13

Extender's Guide for Artifacts and Drafts

Michael Karr
Glenn Holloway
Steve Rozen
~~[illegible]~~ 85

February 8, 1994

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

This document is an extender's guide for those wishing to add new artifact and/or draft functionality to E-L.

Contents

1 Introduction 1

2 User Interface Extensions for Types 1
2.1 The **types** Plex 1
2.2 Doing It, Repairing It, Undoing It 1
2.3 Representation of Artifacts 2
2.4 Resident and Auxiliary Files 4
2.5 Standard Behavior 5
2.6 Simple Use of Templates 7
2.7 Bodies and Contents 8
2.8 Advanced Use of Templates 10
2.9 Advanced EMACS Lisp Extensions for Artifacts 13
2.9.1 Functions Involving Uids 13
2.9.2 Display Functions 14
2.9.3 Other Extension Functions 14
2.10 **diffing** Artifacts 15
2.11 Drafts 16
2.12 Examining **error** Derivatives 22

3 Derivative Class Extensions 1
3.1 Equivalence of Derivatives 1
3.2 The **derivative-classes** Plex 2
3.3 Representation of Derivatives 4
3.4 Fancy Techniques for Representing Derivatives 8
3.5 **error** Derivatives 9
3.6 **universal** Artifacts 10

4 Derivers 1
4.1 How Jobs are Scheduled 1
4.2 The **hosts** plex 3
4.3 The **workers** Plex 4
4.4 The **derivers** Plex 6
4.5 Writing a Deriver 7
4.5.1 Reporting Errors 11
4.6 Language-Independent Utilities 13
4.7 How to Write a Deriver in C 14
4.7.1 How to Create or Extend a C Worker Executable 14
4.7.2 Types and Calling Conventions 16
4.7.3 Relevant Utilities 18
4.7.4 How To Install A Worker Developed in E-L 22
4.7.5 Debugging A Deriver 22
4.8 How to Write a Deriver in Common Lisp 25

4.8.1 How to Create or Extend a Lisp Worker Executable 25
4.8.2 Types and Calling Conventions 27
4.8.3 Relevant Utilities 29
4.8.4 Debugging A Deriver 30
4.9 How to Write a Deriver in EMACS LISP 31
4.9.1 How to Create or Extend an EMACS Worker Executable 31
4.9.2 Types 33
4.9.3 Relevant Utilities 34
4.9.4 Debugging A Deriver 34

A Thumbnail Sketches of Umpteen Derivers **1**
A.1 t, `caption` for Many Types t 1
A.2 t, `manuscript` for Many Types t 1
A.3 `latex-piece`, `manuscript` 2
A.4 `latex-root`, `latex-directory` 3
A.5 `shellscript`, k for $k \neq$ `manuscript` 5

1 Introduction

To extend artifact and/or draft functionality in E-L, you must first have a working knowledge of how E-L is used. E-L is extended from within, by using the system to develop new functionality and then to augment the plexes that describe the structure of the artifacts system.

This guide assumes you are familiar with the *E-L Users' Manual.* To add a tool to the system, you will also need to know how to develop C or Common Lisp programs using E-L, so you should also have read either the *C and Unix Artifacts Manual* or the *Common Lisp Artifacts Manual.*

Artifact and draft behavior. There are several ways to extend artifact and draft functionality.

- Add a new artifact type.
- Add a new derivative class.
- Add a new tool, a *deriver*, for computing a derivative (or set of derivatives) from an artifact.
- Add a new tool, a *non-deriver* that uses derivatives, but does not directly compute them (e.g., print or export to the file system).

The first two are quite independent avenues of extension; the third is usually an adjunct of one or both of the others, but not necessarily; the fourth is usually an adjunct of the second. A new artifact type gives users a new way to record primary information (such as a source program or a manuscript markup) for consumption by tools. A new derivative class gives tools a new way to record derived information for exchange with other tools and for consumption by users. A good extender will focus on reusability of artifact representations and of derivative classes. Tools whose inputs are derivatives from a broadly applicable class can be used without modification as new types are added.

The aspects of extension enumerated above are the subjects of the remaining chapters of this guide. The rest of this introduction is meant to motivate those chapters for first-time extenders.

Adding a type. Different types of artifacts are structured differently and they present themselves quite differently in the user interface. These differences carry over into the persistent representations of artifacts and drafts as well. The variation is achieved by a set of extension facilities in the Epoch-based user interface of the system, which is implemented in EMACS Lisp. These facilities are structured to make common extensions easy with little or no programming, while allowing nitty-gritty control when it's needed. There is, for example, a connection to a template-driven utility for representation translation that simplifies handling of structured artifact headers and other such forms. Among other advantages, this makes it easy to do some syntax checking before a draft is committed, so that drafts without the right gross structure don't even become artifacts.

Adding a derivative class. Technically, a *derivative* is a string associated with an artifact; it contains or it points to information of a particular *kind* about the artifact. Often, a derivative includes the name of a file, relative to a repository subdirectory that holds derived information of that kind. For example, `c-module` artifacts have derivatives of kind `c-file`. A typical such derivative is the string `142.c`, which names a file in the repository subdirectory for `c-file`'s. That file contains C code that has been extracted from the module and its references and put into a form suitable for compilation.

Derivative kinds fall into categories, called *derivative classes*. The members of a class share the same methods for operations like display (via the `examine-derivative` command) and deletion (via `delete-derivative`, for instance). An example of a derivative class is `object`, the class of kinds covering compiled C object files. The kinds in class `object` have names like `object%sun3` and `object%mips-profile` because different target machine architectures and different compilation options produce different kinds of object files. Every derivative kind must have a class, even if it's a singleton whose name is the same as the kind it contains. (The kind `c-file` is the lone member of the class `c-file`.) You add a class by registering it in the `derivative-classes` plex, described in chapter 3.

Tools and Workers. This document assumes that as a user of the current E-L implementation, you have acquired a working understanding of how background processing is managed. You know, for example, that background tasks are called *jobs*. You know about the *scheduling plexes* called `past`, `present`, and `future`, which describe jobs already completed, jobs currently in progress, and jobs waiting to be started. You are at least vaguely aware that the `present` plex identifies processes called *job servers* that appear to take jobs (albeit selectively) from the `future` plex, work on them for a while, and move them into the `past` plex.

A job is based on a *tool*, which names the function that is run in the background. Extensions to the user interface, by contrast, are called *commands*. Tools are divided into two categories, those that compute a derivative (or set of derivatives) from an artifact, called *derivers*, and *non-derivers*. E-L provides tool writers with a simple interface that allows them to ignore the problems of scheduling jobs, locking and reading plexes, and so on. Although a tool is activated by a job server process, it's not actually linked into the job server executable. Tools are collected in executable files called *workers*. You may either create a new worker for a new tool that you write, or plug your tool into an existing worker. Problems like communicating with the job server, handling internal errors, and managing heap storage are taken care of by a nucleus common to all workers in a given implementation language. A worker process is normally spawned as an inferior process by a job server the first time it needs to call one of the worker's tools. But to debug a tool, you can reverse the situation by starting a worker process and then giving it its own job server to act as interface to the artifacts system.

Workers and derivers are registered in plexes called `workers` and `derivers`. These are used by the scheduling algorithm to determine the derivers that can carry out a given job, to identify workers that contain those tools, and then to select an available job server that can run one of those workers. Non-derivers will be discussed later.

Adding a new deriver. In the ultimate E-L system, the developer of a program or a document will work entirely through a single interface that has the feel of an editor, but an editor that can interpret programs and typeset documents. That's why E-L avoids commands for running compilers, text formatters, and the like. The user focuses on creating and manipulating the structure of primary objects, like programs and documents. A *derivation*, that is a job whose tool is a deriver, e.g., for compilation or typesetting, occurs in the background, in response to the creation of new artifacts, and in anticipation of the user's need for derived information. Such processing is only loosely coupled—via plexes—to user sessions. It is distributed on the network to take advantage of available computing power and to avoid degrading session responsiveness.

You may have the idea that derivations are scheduled only when drafts are committed. Actually, commitment and scheduling are completely separate notions; you can commit drafts without scheduling any jobs and schedule deriver jobs without committing any drafts. In fact, new deriver jobs can also be scheduled as part of carrying out existing jobs. While there is only one scheduling algorithm, there is no central scheduling agent. Scheduling is carried out by multiple processes, distributed over the network, cooperating via the scheduling plexes and using still other plexes to guide decisions.

An important aspect of writing a deriver is making it "incremental", i.e., making it efficiently use information derived from similar artifacts. This involves looking at two related artifacts, i.e., two *relatives*, figuring out just what the differences are and what information in the derivative of one is of use in constructing a derivative of the other. Don't be daunted by the apparent complexity of this situation. There is a style common to all derivers, and the addition of a deriver actually consists of the implementation of a handful of relatively simple functions, each of which plays a specific role in determining prerequisites for the derivation, for examining relatives, and for putting the pieces together. Chapter 4 explains how to create a new deriver, incorporate it into a worker, add the deriver to the `derivers` plex, and add the worker to the `workers` plex.

Documentation Conventions. It is common for a single document to describe the implementation of several related artifact types and derivative classes. The first chapter of such a document should be entitled "Public View", and should be divided into two sections, the first describing derivative classes, and the second, artifact types. In both cases, it should describe what other extenders need to know about the data structures in order to write tools. The first of these sections is particularly important, because it tells other extenders how to produce output that can be consumed by tools that you are providing.

Summary. The reason that an E-L user can extend the artifacts system is that much of the behavior isn't hidden or hard-wired. Instead it is captured in plexes that users can change in controlled ways. The programs that implement the behavior of artifacts and drafts, including commands in the user interface and tools that derive and manipulate information about artifacts, are all recorded in plexes that users can modify. This rest of this guide is about the structure of those plexes, the nature of the functionality that the extender can or must add, and the commands in E-L for doing that.

2 User Interface Extensions for Types

This chapter describes how to define a new type for artifacts and drafts, and how to extend EMACS for artifacts and drafts of that type.

2.1 The types Plex

The **types** plex lists all the types in the system. An entry in this plex has the form:

- *name resident-file;version auxiliary-file;version*

In a view on the **types** plex with the format **extenders**, each entry occupies two lines; otherwise, lines are invariably too long. The other possible format—and the default—is **users** in which only the names of the types are listed, in alphabetical order. The order in which types appear is not particularly important. However, various internal data structures use the *index of a type*, i.e., its position in the plex, and it is important that this index not change. If a type other than the last is deleted, you will see, in the extender's view on this plex, a gap (single blank line), created so that remaining types will have unchanged indices. It is thus possible to view the plex and determine the index of a type. When you define a new type, it will occupy the first available gap, if there is any.

The files in the plex are those from which the functionality was installed. They are *not* the actual files used by the system. Rather, copies of the files are made and stored internally, and the system uses these internal copies. The version numbers indicate how many times the type has been "repaired", i.e., the number of times a new file has been installed. A version number is not incremented if the new file is identical to the internal file.

At the beginning of a session, E-L loads all the resident files (i.e., the internal copies), in the order in which the types are listed. It is important to keep resident files as small as possible; this will be discussed in more detail below. The auxiliary files (i.e., the internal copies) are loaded as needed.

2.2 Doing It, Repairing It, Undoing It

We begin with the EMACS command that does the actual installation:

- `new-type` *type resident-file auxiliary-file* `&optional` *initialization-file*
  - The *type* argument is the name of the new type; it may contain only alphanumerics and hyphens.
  - The two files are EMACS lisp that specifies the functions associated with the new *type*. The *auxiliary-file* should end with `(provide 'types-`*type*`)`. These files are discussed in more detail in section 2.4.
  - The resident and auxiliary files will be loaded to check for errors that arise in loading; if such errors arise, the installation of the *type* will not occur. Otherwise, a new entry is made in the types plex.

  - If the *initialization-file* is not `nil`, it is loaded after the *type* is installed and before any startup functions (see **`eval-when-startup`**, below) are called. Its purpose is to initialize the *type*. When **`new-type`** is used interactively, a prefix argument will cause this argument to be requested; otherwise, it will be `nil`.
  - After issuing this command, you will probably want to change the filter of your views on the **`artifacts`** plex so that you can see artifacts with your new type.

If your implementation is correct the first time, you may skip the next function. If however, you discover bugs in the functions for a type, you may install the fixes with the following command, legal only when the type already exists:

- **`repair-type`** *type resident-file auxiliary-file* **`&optional`** *repair-file*
  - The arguments are the same as to **`new-type`** (with the *repair-file* acting like the *initialization-file*).
  - Both files will be loaded; if an error occurs, the repair will not be made.
  - Otherwise, the *resident-file* and *auxiliary-file* arguments supersede those that are currently in the **`types`** plex, but the version numbers are incremented only if the files are actually different.
  - If the *repair-file* is not `nil`, it is loaded after *resident-* and *auxiliary-file*s have replaced the old values. Its purpose is to repair the state of the *type*, for example, *to uniformly change the bodies of artifacts with type type.*

A type may be deleted by the following command:

- **`delete-type`** *type*

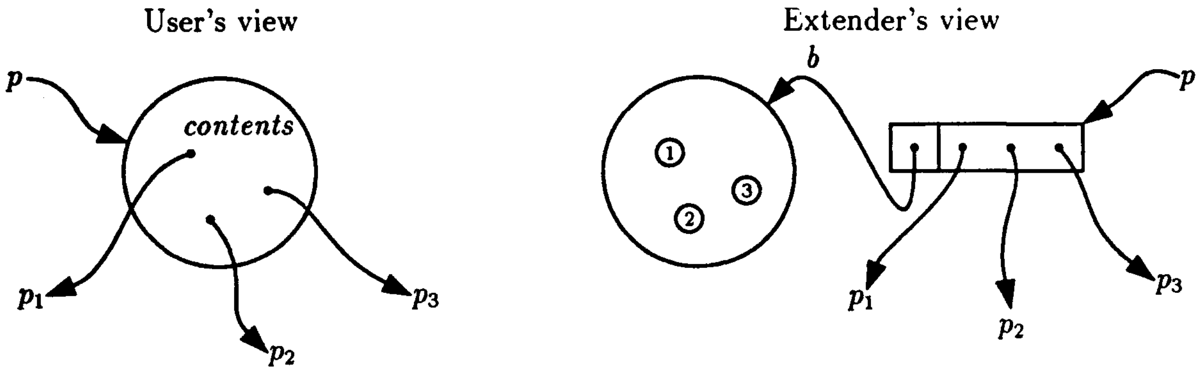
This will not succeed if there are still artifacts with type *type*.

2.3 Representation of Artifacts

To write editor functions and other tools that deal with the internal structure of artifacts, you must understand how E-L stores artifacts. This section describes the storage technique, and is thus a requirement for the remainder of this chapter, and for the next chapter.

From the user's point of view, an artifact consists of a structure—the *contents* of the artifact—determined by the type of the artifact. The contents may have occasional references to other artifacts (see the picture below). The contents may be relatively unstructured, e.g., English text, or highly structured, e.g., base E-L. We are concerned here only with managing the references to other artifacts, not the details of the contents (which are type-dependent).

From the extender's point of view, an artifact is dealt with via an *artifact pointer* (in the picture below, the pointers are p, p_1, p_2, and p_3). In both EMACS Lisp and Common Lisp, an artifact pointer is simply a short string. However, life for the extender is more complicated than simply having artifacts with pointers to other artifacts. A practical implementation of E-L's updating philosophy requires that two artifacts having the same contents, and differing only in references to other artifacts, be storable in only a little more space than is required by one of the artifacts. Accordingly, there is a two-level storage of artifacts, connected by a notion of an artifact *handle*, which is a part of the contents that identifies a reference *relative* to the set of references for the artifact. In the following picture, the handles are shown as circled integers.



While the user has only the idea of contents, an extender must carefully distinguish between what is technically the *body*, labeled *b* above, and the contents, i.e., whatever is inside the circle. In the implementation, a body is again a string. The interpretation of this string is totally type-dependent, and is governed by the following rules:

- The body must not contain a $\boxed{}$, $\boxed{t}$, or $\boxed{n}$.
- Pragmatically, the implementer should ensure that the string is reasonably short. For this reason, it is often essentially a file name (see section 2.9 for the whole story here) where the contents can be found.

The rules governing artifact pointers, bodies, and handles are summarized as follows:

- The contents of an artifact never contains a pointer to another artifact. Rather, at each place where the user thinks there is a reference, there is actually an artifact handle. Handles may be thought of as being numbered consecutively: $1 \ldots k$. However, the representation of a handle is entirely up to the type designer; the artifacts system never looks directly at an artifact's contents.
- The database associates with an artifact pointer a body, a list of references (i.e., a list of artifact pointers corresponding to the handles in the body), a set of predecessors (i.e., their pointers), and a list of universal attributes.

An important consequence of these rules is that even though two artifacts have different references, they can still share a single body. The only cost for the second artifact is the record in the database that associates the pointer, body, references, predecessors, and attributes.

A tool can query the database for the information associated with the pointer, and only type-specific tools know about the structure of contents, so only such tools can associate handles with artifacts. The database knows nothing about the representation of bodies—it treats them only as strings. A tool is nearly as ignorant of artifact pointers—it passes them around but never looks inside them.

2.4 Resident and Auxiliary Files

Because the resident file is loaded at the beginning of an E-L session, it is useful to keep it small, on the theory that the functionality for any given type is likely not be used in a given E-L session. The usual custom is to place in the resident file only user commands and their key bindings. Sometimes, even many of the user commands need not be resident. For example, there may be several commands associated with a debugger, none of which make any sense until the debugger is started. Thus the command to start debugging might be resident, but the other commands can be in the auxiliary file. Further, utilities defined in one plex's auxiliary file and used by another plex should not be made resident, even by an **`autoload`**. The preferred style is to place a call on the following function in the auxiliary file of the plex that may need it. It is also used to define commands which must be visible in the resident file.

- **`autoload-type`** *symbol type string boolean*
  - The arguments are exactly the same as ordinary EMACS Lisp function **`autoload`**, except that a *type* rather than a file is the second argument. The full definition is expected to be in the auxiliary file for the type.

We will also see that E-L will look for certain functions in the auxiliary file even without **`autoload-type`** (see section 2.5). Thus, a resident file can actually be empty, although the is unlikely, because a type will probably have a user command to modify it, and **`autoload-type`** declarations for this should be in the resident file. Similarly an auxiliary file can contain only the **`provide`**, but it is not likely that a type would have no functionality. To repeat a previous point, it is *not* good form to put the implementation for a type in the resident file, thereby making the auxiliary file minimal.

In some cases, the auxiliary functions for one type may use **`autoload-type`** to define functions from another type. An auxiliary file may also use:

- **`require-type`** *type*
  - Same as the usual EMACS **`require`**, but takes a string argument and implicitly specifies the auxiliary file for *type* as the file to be loaded.
  - It does not make sense to use this in a resident file, because that would have the effect of making the auxiliary file for *type* resident.

It is also possible to have EMACS-lisp files not associated with any particular type. For this, use one of:

- **`require-emacs-lisp`** *file*
  - The argument is a string that specifies a *file* known to E-L as the one to be present.
  - Such a file should end with `(provide '`*file*`)`.
  - This is occasionally used in resident files to make basic functionality resident. Such uses should be rare and conscious.

- **autoload-emacs-lisp** *symbol file string boolean*
  - Same as **autoload**, but the *file* is one that is known to E-L.
  - Utilities are described in the document *E-L Emacs Utilities.* To determine the *file* argument for this command, see the italicized tag that is near the right margin in the implementation of the utility.

To make the loading of resident and auxiliary files idempotent, any initialization of data structures should not be done directly by top-level forms in the file, but within "startup declarations", specified by:

- **eval-when-startup** *forms*
  - The forms are recorded rather than evaluated when loading. They are evaluated after the load is complete, and then only the first time a file is loaded.
  - Startup declarations are evaluated in the order in which they occur in the file. (The resident files for installed types are loaded in the order in which types are installed; auxiliary files may be loaded in any order.)

Finally, a note on successful use of the **repair-type** capability: loading a resident or auxiliary file should be idempotent. Thus, it is acceptable to use **defconst** to define constants (perhaps even preferable, if those constants are subject to being defined wrong), but **defvar** should be used for variables that hold session state. One of the purposes of **eval-when-startup** is to ease fulfilling the idempotency requirement.

2.5 Standard Behavior

There is a set of type-dependent EMACS functions and variables that are subject to customization. An extender may define none of these, or any subset of them, as long as the definitions are self-consistent. If *f* is a "family" of type-dependent functions or variables, the user customizes *f* for type *t* merely by defining a function or variable whose name is *f*-*t*, e.g., (**def**[un|var|const] *f*-*t* ...). If the extender makes no such definition, the function or variable **standard-***f* is used instead. Moreover, if this function is not resident, the system will look in the auxiliary file for the type *t* (see section 2.2 and 2.4), so there is no need to make these functions resident even with an **autoload-type**. To keep resident functionality small, it is in fact preferable to say nothing about *f*-*t* in the resident file for *t*.

The functions **standard-***f* are also available to the extender. The first argument of **standard-***f* is a type (a string, sometimes ignored), and the subsequent arguments are the same as *f*-*t*. It is common for an implementation of *f*-*t* to call **standard-***f*. In many situations, this is recommended:

- If **standard-***f* is not entirely trivial, then it is good practice for *f*-*t* to call **standard-***f*, and to add its own functionality by other actions before and/or after its call. In particular, it is recommended that you not find the implementation of **standard-***f* and copy its pieces into your function, because this makes it less likely that your extension will pick up improvements made to standard functions and less likely that they will be robust in the face changes to the implmentation details of standard functions.

The standard way to store the contents of an artifact of type **type** is as a file in the directory *repository*/**types**/*type*, where *repository* is the directory in which data for E-L is stored. The standard body for an artifact of type *t* is a file name relative to the standard directory. The file name is automatically generated, and is usually referred to as a *uid*, for "unique identifier". A uid is represented as a string.

The contents of an artifact is standardly represented as *standard artifact text*, which is simply ordinary text with the following convention for dealing with handles:

- The i^{th} handle (1-based) is represented by the character code $256 - i$.

This means that there is a limit of 128 handles per artifact buffer before these characters start to overlap with standard ASCII. [This is subject to future extension; for the present, there are pragmatic reasons to avoid artifacts with this many references.]

There is in general a difference between the contents of an artifact as stored in the repository (*repository form*), which is what was described above, and what is reasonable for a person to look at (*display form*), in which case, handles are represented in one of two ways, either by special characters (for non-Epoch use), just as in standard artifact text, or as Epoch buttons. Clients of this facility thus may make no assumptions about the representation, but must use utilities for various operations. Drafts often use the display form of standard artifact text.

If standard artifact text, both in display and repository form, is sufficient for the purposes of your extension, the only definitions that you will need to make are those that specify edit modes and that inform the system about derivative kinds for an artifact type:[1]

o **artifact-edit-mode-***type*

- E-L calls this with the current buffer having the repository form of an artifact of the given *type*.
- Since buffers on artifacts are read only, there is not a lot for this function to do, but it is often useful to set the edit mode for better navigation in the buffer, e.g., so that `C-x C-f`, etc., do the right thing for Lisp artifacts.
- **standard-artifact-edit-mode** makes artifact references interact with the kill-ring by pushing a suitable function onto **copy-buttons-as-kill-hooks**.

o **derivative-kinds-***type body* ⟶ *kind-&-priority--list*

- This must yield a list of pairs consisting of the possible kind of derivatives (see chapter 3) for an artifact of the given *type* having the given *body*, together with the default priority for scheduling jobs to obtain the derivatives. A priority is number between 1 and 9, with lower numbers having higher priority, or else is **nil**, indicating that no job should be scheduled. The usual priority is 5 or **nil**.

[1]The "hollow bullets" indicate functions that an extender must write. Solid bullets, if they introduce functions, are used when the function is built-in and available for the extender.

- *All* of the possible kinds must occur, even if they are never to be derived opportunistically, because this list is also used in deleting derivatives when an artifact is deleted.
- In many but not all cases, the result is independent of the *body*.
- **`standard-derivative-kinds`** yields the null list.

If defined, the definitions of these functions, like all functions an extender defines, go in the auxiliary file.

2.6 Simple Use of Templates

This section requires knowledge of the public operations on templates, which are discussed in *E-L* EMACS *Utilities*. There is no point in continuing until that material is understood.

We saw in the previous section that if you are happy with standard artifact text, the only functions you need to write for a type extension are for edit modes. In this section, we provide a means of extending EMACS lisp functionality that is an intermediate point between the strait-jacket and simplicity of standard artifact text and the flexibility and complexity of arbitrary extensions. It covers situations in which the difference between the repository and display forms for artifacts of the given type can be characterized by two templates that describe the first part of the contents of the artifact in the respective forms:

- To obtain the display form, extract a value with the "repository template", delete the matched text, and re-insert the value with the "display template". Set the buffer-local variable **`header-end-point`** to a marker just after the inserted text.
- To obtain the repository form, do the same thing, but exchanging the two templates, and leaving **`header-end-point`** unaffected.

For this to work, it is clearly necessary that the values associated with the two templates have the same structure, i.e., that whatever is extracted with one can be inserted with the other.

A simple example occurs in `lisp-source` artifacts. In display form, a buffer begins with `Caption`: (in inverse-video and read-only) and the rest of the first line is the text of the caption. In repository form, the `Caption`: is replaced by a simple semicolon, for the expedient reason that the **`lisp-forms`** derivative of the artifact can be shared with the body (at least when there are no references, which is the usual case).

To obtain this simple correspondence between display and repository form, it is necessary to ensure that the template machinery is loaded and to declare that the type in question is using templates, and that is all. For the `lisp-source` example, the following declaration goes in the auxiliary file for `lisp-source` artifact. (See below for the syntax-category `caption`.)

```
(require-emacs-lisp "templated-headers")

(use-templates "lisp-source" '(concat caption line) '(concat ";" line))
```

The first argument here is the type. The other arguments are template specifiers, as discussed in *E-L* EMACS *Utilities.* All that needs to be understood here is that templates provide a way of mapping between text in a buffer and EMACS lisp values; use of the functions for templates is not required. The above example is an instance of:

- `use-templates` *type template-specifier template-specifier*
  - The second argument specifies the display form for the header of artifact contents of the given type, and the third argument specifies the header for the repository form.
  - In the absence of any further declarations (see section 2.8), the values associated with the two *template-specifiers* must have the same structure, and the part of the body not specified by the template is standard artifact text.

For applications of templates in artifacts, the EMACS Lisp utilities `templated-headers` (example above) does a `(require-emacs-lisp "templates")`, and in addition, defines the following template syntactic categories that are specific to artifacts and drafts.

- `handle`—the syntax is the standard display of an artifact reference. The data is an integer, specifically, a 1-based handle index. (Do not use this for repository form! It sets up the *display* for a handle in text form.)
- `caption`—this syntax category is effectively `(permanent "Caption: ")`.

Uses of the `handle` category are often in conjunction with the machinery provided in the next section, and we postpone an example.

Recall the discussion in section 2.5 of the customization of a family *f* for a type *t*. The way that `use-templates` works is to make a stylized definition of `f-t` for all families *f* where template-controlled behavior is desirable. These definitions all have the form:

- `(defun f-t (&rest args) (apply 'templated-f t args))`

Any definitions of `f-t` that follow `use-templates` will override this canonical definition; it is thus possible to get the templated behavior as a default, but do further customization where necessary. Such further customization will often use a direct call on `templated-f` to do most of the work. The behavior of `templated-f` will be given as part of the descriptions of each family, in later sections.

2.7 Bodies and Contents

As noted above, the contents of an artifact is usually stored in a file named by a uid. It is possible to make most extensions without having to know where these files are stored or how the uids are obtained. In fact, as we saw in the last section, it is often possible to make extensions without even being aware of what a uid is. In those extensions, the body of the artifact is exactly the uid.

It is quite common to customize artifact bodies of a given type by including in the body a bit more information than just the uid. The motivation is usually speed in finding out something about the contents without having to do a file access. An example is a body that has both a uid and an integer. The standard values of the following provide the case in which a body is exactly a uid.

- `uid-of-artifact-body-re-format-`*type*
  - This is a regular expression with a "hole" that locates the uid within the body. The hole is indicated by `%s`. For example, if the body consists of a uid followed by a vertical bar and an integer, the string should be `%s|[0-9]+`. (See section 2.8 for an example.)
  - NB: There must not be a `\(` preceding the hole.
  - If this value is the empty string, then it is assumed that the body itself is the entire contents of the artifact.
  - `standard-uid-of-artifact-body-re-format` has the value `%s`, i.e., the uid constitutes the entire body.
- `artifact-body-uid-re-`*type*
  - This is a regular expression that must match a uid.
  - `standard-artifact-body-uid-re` has the value `[0-9]+`, i.e., a standard uid has one or more digits.

To be completely precise about how these two variables work, we describe how they are used. One issue is the deletion algorithm, which must cope with the fact that many artifacts can share the same body (see section 2.3), and thus the same uid, which may not be deleted until the last reference to it has disappeared. Briefly, after an artifact is deleted, the following algorithm is applied to its body.

- If `uid-of-artifact-body-re-format-`*type* is the empty string, there is no uid, and nothing else to do (`delete-artifact-uid-`*type* is never called). Otherwise, the body does have a uid.

- Denote `artifact-body-uid-re-`*type* by *uid-re*, let *body-re* be the result of substituting `\(`*uid-re*`\)` for the hole in `uid-of-artifact-body-re-format-`*type*, and let *re* be `^`*body-re*`$`. Then *re* must match the entire string (that is the role of `^` and `$`), and the uid is extracted as the first match. This construct explains why `uid-of-artifact-body-re-format-`*type* must not have a `\(` preceding the hole.

- Another artifact is said to "share a uid" with the deleted artifact if its body is exactly matched by the regular expression obtained by substituting the uid for the hole in `uid-of-artifact-body-re-format-`*type*. A search is made for artifacts with the same type and sharing the same uid. If none are found, `delete-artifact-uid-`*type* is applied to the *uid*.

Another situation in which these same variables are used is in archiving. In this case, **uid-of-artifact-body-re-format-***type* is used to determine if the artifact has a uid; if so, its contents become part of the exported archive, and when importing the archive, a new uid is created using them, and a new body constructed from the non-uid part of the archived artifact's body and the new uid.

2.8 Advanced Use of Templates

The use of templates alone is insufficient to implement a design in which the body of an artifact has information that is in the buffer in display form. This section provides a paradigm for such situations. It assumes knowledge of built-in template syntax categories and constructors (see *E-L* EMACS *Utilities*) as well as ordinary EMACS regular expressions.

The first step is to define the templates. In the example below, it is clear that the two templates are not inverses of one another; they don't even contain the same information. The list of handles in the first template disappear in the second. In repository form, rather than being stored in the contents of the uid, it will be represented as an integer, which together with the uid, constitutes the body. The integer, say n, says that the first n references are part of the header; the remaining references are found in the contents of the uid, stored as in standard artifact text.

```
(require-emacs-lisp "templated-headers")

(use-templates "module"
  '(concat* caption line
            (permanent (divider "Prerequisites"))
            (list (concat handle "\n"))
            (permanent (divider "")))
  'line)
```

The display forms looks like:

```
Caption: caption
------ Prerequisites --------------------------------------
reference_1
...
reference_k
-----------------------------------------------------------
```

The repository form has only the *caption*; as we shall see, the references are stored differently.

When the repository and display templates have associated values that are structurally dissimilar, it is necessary to define three functions used in converting between the two forms. The function that produces display form gets as an argument the body with the uid stripped out, and produces the values to be inserted via the display template. There are two functions used to produce repository form from the display form of a buffer. The first yields a "body format" and a list of handles. The second produces the values to be inserted via the repository format. The standard versions of these functions give the behavior described in section 2.6.

o **`templated-repository-to-display-`***type residual-body w* → *v*

- Called on a buffer with the repository form of an artifact, after **`standard-display-artifact-contents`** has been called. It must not disturb point.
- The *residual-body* is called that because it is the body with its uid removed. Technically, it is obtained by using the value *re* defined in connection with deletion (see the previous section) to locate the uid within the body, and to yield the concatenation of the strings preceding and following the uid substring. With a standard body format, namely `%s`, the *residual-body* is always the empty string.
- The value *w* was obtained by using the repository template to parse the buffer, starting at the beginning.
- The value *v* will be inserted according to the display template.
- This function is relevant only when **`use-templates`** is in effect.
- **`standard-templated-repository-to-display`** checks that the residual body is the empty string (issues an error if not), and yields *w* as the value for *v*.

o **`body-format-&-handles-that-appear-in-header-`***type v* → *body-format handle-list*

- The value *v* passed to this function was extracted from the buffer using the display template; the *point* parameter was the end of the match.
- The first result is a *body-format*, a string with a "hole" (`%s`) in it for the uid. The *body-format* becomes the body after the hole is replaced by the uid, unless `uid-of-artifact-body-re-format-`*type* is the empty string, in which case the *body-format* is the body itself.
- The last result is a list of handles actually appearing in the part of the buffer matched by the display template. The references of the eventual artifact will first have these references in the order specified, and then those of the rest of the buffer, in order.
- The two results must be combined with **`cons`**.
- This function is relevant only when **`use-templates`** is in effect.
- **`standard-body-format-&-handles-that-appear-in-header`** yields a `%s` as the body format, and a null list of handles.

o **`templated-display-to-repository-`***type point v* → *w*

- Called when the current buffer is on a buffer with the display form of an artifact, or a buffer on a draft. It must not disturb point.
- The value *v* and *point* are the same as in the above function.
- If this function returns without error, *w* will be inserted according to the repository template, and if no error occurs, the text matching the display template will be deleted.

- When this function is used in a commit, signaling an error will abort the commit.
- This function is relevant only when **use-templates** is in effect.
- **standard-templated-display-to-repository** yields v as the value for w.

Definitions of these functions for the example of this section, and the variables describing the structure of the body are given below. (NB: The `%%` in the argument to **format** below results in a single `%` in its result; the `%d` is replaced by the decimal digits of the length of the list of handles. Thus, the result is a legitimate body-format.)

```
(defun templated-repository-to-display-module (residual-body caption)
  (let ((n (string-to-int (get-match "|\\([0-9]+\\)" residual-body)))
        (handle-list nil))
    (while (> n 0)
      (ppush n 'handle-list)
      (decr 'n))
    (cons caption handle-list)))

(defun body-format-&-handles-that-appear-in-header-module (caption-&-handle-list)
  (let ((handle-list (cdr caption-&-handle-list)))
    (cons (format "%%s|%d" (length handle-list)) handle-list)))

(defun templated-display-to-repository-module (end caption-&-handle-list)
  (car caption-&-handle-list))

(defvar uid-of-artifact-body-re-format-module "%s|[0-9]+")
```

See *E-L* EMACS *Utilities* for **get-match**, **decr**, and **ppush**.

As another example of the use of these functions, we show how they are used in the implementation of `lisp-source` artifacts—the description in section 2.6 was an oversimplification. The real story is this: if the caption in display form is not blank, the caption is repository form is "`;1`" followed by the caption text supplied by the user. If the caption in display form is blank, the caption in repository form is "`;2`" followed by a caption that is automatically filled in by EMACS. (A caption of the latter form is not typeset with the **source** artifact itself, only at references to it.) So in addition to the templates described earlier, there are also definitions of two of the three possible functions. (The missing one is not needed because bodies in this case are standard.)

```
(defun templated-repository-to-display-lisp-source (residual-body caption)
  (if (= (aref caption 0) ?1)
      (string-cdr caption)
    ""))

(defun templated-display-to-repository-lisp-source (point caption)
  (if (string= caption "")
      (save-excursion [code which constructs the default caption])
    (concat "1" caption)))
```

See *E-L* EMACS *Utilities* for `string-cdr`.

2.9 Advanced EMACS Lisp Extensions for Artifacts

This section has the full story on extending EMACS functionality for an artifact type. It is required reading only when standard artifact text and templates are not satisfactory for your extension.

2.9.1 Functions Involving Uids

You can override the default use of the artifact repository, and put artifacts of a given type wherever you want.

o **`new-artifact-uid-`***type* → *uid*

- Given no arguments, this function must produce a string without $\boxed{\ }$, $\boxed{t}$, or $\boxed{n}$ that is a suitable uid for artifacts of the given type.
- **`standard-new-artifact-uid`** uses the file *repository*/`types`/*type*/**`serial-number`** to yield a uid (not a full pathname) for a heretofore non-existent file or subdirectory in the *repository*/`types`/*type*/ directory.

o **`write-artifact-contents-`***type uid*

- E-L calls this function with a *uid* obtained with the previous function. It is to write the contents of the current buffer to the *uid.*
- **`standard-write-artifact-contents`** writes the current buffer to the file *repository*/`types`/*type*/*uid.*

o **`read-artifact-contents-`***type uid*

- E-L calls this function with a *uid* to which the previous function has been applied, and the following one hasn't. It is to read what was written into the current buffer after point.
- **`standard-read-artifact-contents`** inserts *repository*/`types`/*type*/*uid* into the current buffer.

o **`delete-artifact-uid-`***type uid* → *boolean*

- E-L calls this function with a *uid* which is no longer referenced. It should scavenge the space now used by the *uid.*
- The result should be `nil` if the *uid* was already deleted; otherwise, `t`. (If two sessions are racing to delete artifacts with the last references to the *uid*, it is possible for them both to attempt to delete it. Thus the `nil` result is rare, but possible.)
- **`standard-delete-artifact-uid`** deletes the file *repository*/`types`/*type*/*uid.*

2.9.2 Display Functions

When the user asks to examine an artifact, the generic function calls **`read-artifact-contents-`***type* to initialize the buffer with the repository form for the contents, then calls the first of the following functions to produce the overall display form, and finally calls the second one for each of the handles.

o **`display-artifact-contents-`***type residual-body*

- The current buffer has been initialized with the contents of an artifact in repository form.
- The *residual-body* is the same as for **`templated-repository-to-display-`***type*.
- In Epoch, **`standard-display-artifact-contents`** covers each meta-character $256 - i$ with a button. In EMACS, the standard function is a no-op.
- **`templated-display-artifact-contents`** initializes the buffer using **`standard-display-artifact-contents`**. It then extracts w using the repository template, dispatches to **`templated-repository-to-display-`***type* to obtain the value v, deletes the matched region, and using the display template, inserts these values into the buffer.

o **`display-artifact-reference-`***type handle artifact-pointer*

- E-L calls this with the current buffer initialized to the repository form of artifact text, modulo any changes made by the previous function and by previous calls on this function. This is first called with the *handle* equal to 1, then 2, etc.
- The first argument is a handle, and the second is the corresponding artifact pointer. This function must arrange for displaying occurrences of the handle in the current buffer as references to the given artifact. [A later edition will describe how to get from an artifact pointer to a string representing the artifact.]
- **`standard-display-artifact-reference`** builds up a list of strings for the artifact references, storing it in the buffer local variable **`special-char-strings`**, anticipating the day when meta-characters can be displayed as arbitrary strings. In Epoch, it replaces the text under the 1-based ith button with ith element of **`special-char-strings`**. In EMACS, there are no changes to the contents of the buffer.

2.9.3 Other Extension Functions

The implementation of an artifact type is expected to provide sensitivity to references from within a buffer on an artifact. Since the generic mechanism assumes nothing about the internal representation of a buffer, it uses:

o **`accept-handle-from-artifact-buffer-`*`type`*** $\rightarrow$ *handle* $\cup \{0\}$

- E-L calls this function when a command requires an artifact argument and point is in a buffer on an artifact of the given *type*. If point is near a reference, for some suitable definition of "near", the result should be the handle for that reference; otherwise, the result should be 0.
- **`standard-accept-handle-from-artifact-buffer`** defines "near" to mean "on the same line". Preference is given to the first reference following point; otherwise the first reference preceding it. In Epoch, it is necessary to look at the button-list; in EMACS, this is done by searching for meta-characters.

Finally:

o **`artifact-commands-`*`type`***

- This specifies the list of all *type*-specific artifact commands, i.e., commands that take an artifact argument whose type is constrained, but is at least allowed to be the given *type*. E-L uses this variable in constructing the menu that appears when mousing an artifact.
- **`standard-artifact-commands`** is **`nil`**.

2.10 diffing Artifacts

An extender may write several functions to specialize the behavior of **`diff-artifacts`** (and related commands) for a new type. Because of the possibility of using such commands on artifacts of differing types, *all* types must interact with diffing via the least common denominator of text in a buffer. First, to obtain the text, the system calls:

o **`diff-input-from-artifact-contents-`*`type`*** *separator string-list buffer*

- The current buffer has the display form of the contents of an artifact.
- The *separator* is a string that does not appear in the buffer, nor in the buffer that this artifact will be diffed against.
- The *string-list*, which we will denote $\langle s_i \rangle_i$, has an element for each handle.
- The job of this function is to copy the current buffer into *buffer* (which is initially empty), and to replace each handle with the string xs_ix, where x is the separator.
- **`standard-diff-input-from-artifact-contents`** expects that the current buffer will have standard artifact text in display form.

When diffing artifacts of the same type, the following function will be called to compute the difference. For example, the extension for a type may choose to diff on a section by section basis.

- **`diff-artifact-`***type buffer buffer buffer* → *delta-list*
  - The first two arguments have been prepared by **`diff-input-from-artifact-contents-`***type* .
  - The *delta-list* data structure is explained in *E-L* EMACS *Utilities.*
  - **`standard-diff-artifacts`** is **`diff-buffer-fn`**; see *E-L* EMACS *Utilities* for details.

When diffing artifacts of different types, the system calls **`standard-diff-artifacts`**.
To display references in the diff result, one of the following two functions is called.

- **`insert-artifact-reference-`***type artifact* {`nil`, **`old`**, **`new`**}
  - Insert the display form of an artifact reference at point, highlighted according to the second argument.
  - **`standard-insert-artifact-reference`** uses white on blue for `nil` (just like standard **`display-artifact-references`**), red on blue for `old`, and green on blue for **`new`**.

- **`insert-two-artifact-references-`***type artifact artifact* → *delta*
  - A single *xsx* corresponds to the given *artifacts* in the respective arguments to a diff function. This function should display both, in the order given, as if there were separate calls to the above functions with a second argument of `old` and **`new`**.
  - The only reason that this is a separate function is so that the extender can choose how to embed the difference references. This should be reflected by the result.
  - **`standard-insert-two-artifact-references`** replicates the line in which the reference occurs, remembering where point is in each. The *delta* specifies the first and second line as the red-green regions. Other artifact references in the line then result in calls on the previous functions.

2.11 Drafts

When you define a type, you must decide whether you wish to allow drafts of that type. If so, you have the option of customizing the standard operations for drafts. This section describes how to do so. Note that for drafts a body *is* interpreted by the system; it must be a legitimate file name with a fully specified path, i.e., what we will call a *file*. All functions below must be defined in, or are utilities written in, EMACS Lisp.

Extenders of functions that deal with references in drafts must be aware of the fact that drafts have handles just like artifacts, but that each handle is associated not with a single artifact (or draft), but with a set of what are called *subreferences*, each subreference referring to either an artifact or a draft, distinguished by a + or - somewhere in the structure of the subreference.

Extenders must also interact with E-L's support for red-green regions in drafts. Much of the functionality for this is type independent, for example, the way in which red-green regions are highlighted or the fact that it is an error to commit a draft with red-green regions. There is however, an opportunity to define functions like those that are used in diffing artifacts.

The representation of a draft in a buffer is usually very close to that of the display form of an artifact of the same type. There is no notion of "repository form" for drafts, although drafts do have an *auto-save form*, as will be seen below.

o **`new-draft-body-`***type* → *file*

- Must produce a file name (including path) for drafts of the given type. This file will become the auto-save file for the buffer.
- This function is allowed to change the contents of the `current-buffer`, for example, to initialize the draft.
- `standard-new-draft-body` produces the string *repository*/`types`/*type*/*uid*, with *uid* chosen so that the string is not currently a file or directory.

o **`create-draft-`***type*

- E-L calls this with no arguments, and expects it to yield no results. However, the current buffer is already set up, and if desired, this function may initialize it in some useful way (but see below if you want artifact references in the initial text).
- This is part of the action taken by user commands that create a draft without predecessors, e.g., `create-draft`.
- `templated-create-draft` inserts the display template, supplying no values, and thus inserting the default values.
- `standard-create-draft` does nothing.

o **`draft-contents-from-artifact-contents-`***type residual-body*

- When E-L calls this, the current buffer has been initialized with the contents of an artifact in repository form. Its job is to make the buffer editable.
- The *residual-body* is the same as for `templated-repository-to-display-`*type*.
- This is part of the action taken by user commands that create a draft with predecessors, e.g., `edit-artifact`.
- `standard-draft-contents-from-artifact-contents` dispatches to the family member `display-artifact-contents-`*type*.

o **`display-draft-reference-`***type handle draft-reference*

- E-L calls this in several circumstances. The first is in initializing a buffer on a draft; after calling `draft-contents-from-artifact-contents-`*type*, E-L calls `display-draft-reference-`*type* once for each handle, in order of increasing handles. The second circumstance is when a reference has been added to a draft buffer. In both these cases, the *handle* will be one larger than the previous largest handle, or 1, if there were no previous handles. The final circumstance is when the reference for a handle changes, in which case the *handle* will already have been seen.

- The second argument is a list of subreferences, each of which is of the form `("+" . ` *artifact-display-list*`)`, or `("-" . ` *d draft-display-list*`)`. [Display lists for artifacts and drafts will be discussed in a later edition.]
- This function should arrange for proper display of all the occurrences of the *handle* in the current buffer, according to the information provided by the second argument.
- In Epoch, **`standard-display-draft-reference`** obtains the text for the handle and displays it in a button. Otherwise, it updates a list of "special-char" strings, which we hope that one day will be used in the display of meta-characters.

o **`draft-edit-mode-`***type*

- This is the last chance to do anything in a type-dependent way before the user starts editing. It is common for this function to establish editing modes.
- **`standard-draft-edit-mode`** calls **`artifact-edit-mode-`***type*, and (for Epoch) makes buttons deletable as long as the deleted region spans the entire button and makes references interact with the kill-ring by pushing suitable functions onto **`copy-buttons-as-kill-hooks`** and **`yank-hooks`**.

o **`insert-handle-`***type handle* {**`nil`**, **`old`**, **`new`**}

- Insert the *handle* at point in its display representation, highlighted according to the second argument.
- In Epoch, **`standard-insert-handle`** sets up a button with appropriate colors, as in **`insert-artifact-reference-`***;* in EMACS, it simply inserts a meta-character.

o **`accept-handle-from-draft-buffer-`***type* → *handle* $\cup$ {0}

- E-L calls this function when a command requires an artifact or draft argument and point is in a draft buffer of the given *type*. If point is near a reference, for some suitable definition of "near", the result should be a handle for that reference: otherwise the result should be 0.
- **`standard-accept-handle-from-draft-buffer`** interprets "near" to mean "on the same line". Preference is given to the first reference following point; otherwise, the first reference preceding it.

- **body-format-&-handles-that-appear-in-draft-***type* → *body-format handle-list*
  - Called when the current buffer is on a draft.
  - The first result is a *body-format*, which is a string with a "hole" (`%s`) in it for the uid, unless **uid-of-artifact-body-re-format-***type* is the empty string, in which case, it is the body itself.
  - The second result is a list of handles actually appearing in the buffer. The *i*th element of this result will correspond to the *i*th reference of an eventual artifact.
  - These results are represented as a cons.
  - **standard-body-format-&-handles-that-appear-in-draft** yields **%s** as the body format, and calls the following function to obtain the *handle-list*.
  - The function **standard-handles-that-appear-in-draft** takes no arguments, and yields a handle list based on the current buffer. In Epoch, it looks for buttons that correspond to handles; in EMACS, it finds meta-characters. In both cases, the order of handles in the result is that in which they first appear in the buffer, with duplicates removed.
  - **templated-body-format-&-handles-that-appear-in-draft**type obtains *v* via the display template. It stashes the the end of the header (as a marker) and *v* on a buffer-local variable, and calls **body-format-&-handles-that-appear-in-header-***type* on *v* to obtain the body format and handles in the header. It yields as results the body format and the concatenation of the handle list from the header and the handles in the rest of the buffer, obtained with **standard-body-format-&-handles-that-appear-in-draft**.

- **artifact-from-draft-lock-&-mode-list-***type* → *lock-&-mode-list*
  - It is very rare to need to extend this function. If you do, it is probably because of the need to examine plexes when committing an artifact.
  - To understand locking in general, and how to create a *lock-&-mode-list* in particular, see *Extender's Guide for the E-L System.*
  - **standard-artifact-contents-from-draft-contents-lock-list** yields a list of one element, used to obtain a new serial number for artifact bodies of the given *type*.

- **artifact-uid-from-draft-contents-***type* *handle-list* **&rest** *goods-list* → *uid*
  - When E-L calls this, the current buffer is a buffer on a draft.
  - It must transform the contents of the draft to the repository form of the contents for an artifact that is about to be committed. Then it must store the contents in the repository and return a uid for incorporation in the new artifact body.
  - If **draft-edit-mode-***type* set up any buffer-local variables that might do future harm (when the buffer becomes a buffer on an artifact), this function should kill them.

- The *handle-list*, call it $\langle h_i \rangle_i$, is the second result of **`body-format-&-handles-that-appear-in-draft-`***type*.
- The *goods-list* has one element for each element in the result of the previous function. If you understand locking, you will know what this is; otherwise, think of it as a magic cookie which will eventually be given to the standard version of this function (see below).
- This function must modify the current buffer so that the draft handle h_i becomes artifact handle i.
- If desired, this function can refuse permission for the commit, simply by using **`error`**. In this case, it should leave the contents of the buffer unchanged.
- **`standard-artifact-uid-from-draft-contents`** calls two functions discussed in the items immediately following this one: **`standard-artifact-contents-from-draft-contents`** and **`standard-new-artifact-uid-from-contents`**.
- **`standard-artifact-contents-from-draft-contents`** takes two arguments, the *type* (ignored) and the list of handles. It replaces references in a draft with meta-characters for the new artifact handles, and kills the buffer-local variables set up by **`standard-draft-edit-mode`**: **`copy-buttons-as-kill-hooks`** and **`yank-hooks`**.
- **`standard-new-artifact-uid-from-contents`** takes two required arguments, the *type* and the magic cookie corresponding to the one element in the result of **`standard-artifact-contents-from-draft-contents-lock-list`**, plus an optional extension (for use when a body must have one). It uses its magic cookie argument to obtain a new uid and writes the buffer to the file obtained from that uid (possibly incorporating the extension).
- **`templated-artifact-contents-from-draft-contents`** takes the same two arguments as the standard function. It uses the data stashed by **`templated-body-format-&-handles-that-appear-draft`** to delete the header (having first deleted any buttons there), canonicalize the handles not in the header, insert *w* using the repository template, obtain a new uid, and write the contents to the file.

o **`merge-artifact-into-draft-`***type body diff-flag*

- When called, the current buffer is on the draft in question.
- The body of the artifact of the given *type* is *body*.
- If *diff-flag* is true, there is supposed to be a diff representation. Otherwise, the contents of *body* should be appended to the draft, for some suitable definition of append.
- **`standard-merge-artifact-into-draft`** does a **`read-artifact-contents-`***type* at the end of the buffer, narrows to the region just read, does a **`draft-contents-from-artifact-contents-`***type*, widens, and places the cursor just before the new material.

  - **templated-merge-artifact-into-draft** inserts the contents of the artifact with **standard-merge-artifact-into-draft**, extracts the header *v* using **display-template-***type*, deletes the header, and calls **merge-header-into-draft-***type* on *v*.
  - **standard-merge-header-into-draft** inserts the **prin1** form of *v* in place of the header. We are also considering having this function signal an error.

o **auto-save-draft-***type buffer buffer*

  - E-L calls this function when autosaving a buffer on a draft and when abandoning a draft. The first *buffer* is the buffer on the draft; the second is a temporary buffer which is to be filled with the data which E-L will write to the auto-save file. The form of the contents of the second *buffer* is the operational definition of the "auto-save form" of a draft of this *type*.
  - The purpose of this function is to allow the extender to auto-save the values of buffer-local variables. It should not modify the contents of the first buffer, because such changes would become part of the undo information for the buffer.
  - **standard-auto-save-draft** appends to the temporary *buffer* the value of point and mark, a description of each Epoch button in the draft, and the contents of the draft *buffer*. In order for buttons to be recoverable during **takeover-draft** or when **revert-buffer** is applied to a draft, their **button-data** must be such that it gives an **equal** value after being **print**ed and **read** back, where **equal**, **print**, and **read** are the EMACS Lisp primitives. Note that functions for non-standard cases may insert whatever information they need in the second buffer and then call **standard-auto-save-draft** to do the rest.

o **revert-draft-***type*

  - This function is a companion to the above, and is used when recovering or taking over a draft. The current buffer has been initialized to the auto-saved data, and is to be restored to the state at the time of auto-saving or abandonment.
  - **standard-revert-draft** sets up point and mark from the contents of the buffer and deletes this information; it extracts and deletes button descriptions, and then recreates the buttons; and it updates artifact- or draft-reference buttons to reflect changes that may have taken place since the draft was auto-saved. Buttons are re-created with the aid of the buffer-local list **revert-button-hooks**. Each member of that list is a function that takes a **button-data** value and yields a button attribute value or else **nil** (meaning "try the next function"). The default initialization of **revert-button-hooks** contains a single function, called **button-attribute-if-reference**, that deals with artifact- and draft-reference buttons. Other functions can be added to the list by **draft-edit-mode-***type*.
  - **templated-revert-draft** first calls **standard-revert-draft**. It then tries to extract the header with the display template, and if it succeeds, it re-inserts the

value with the same template. Observe that autosave cannot save the draft in repository form, because there is no guarantee that the header has correct syntax in the middle of an edit session. The extraction and re-insertion techniques works most of the time, and has the side-effect of nicely adapting dividers to new window sizes. Even when it doesn't work, no harm is done, because the user can fix the header and proceed.

Suppose, for example, that you want auto-saving to record the value of a string-valued buffer-local variable `this-syntax`, for type `source`. This would be done using the standard versions of these functions, as follows:

```
(defun auto-save-draft-source (draft-buffer temp-buffer)
  (set-buffer draft-buffer)
  (let ((syntax this-syntax))
    (set-buffer temp-buffer)
    (insert string))
  (standard-auto-save-draft "source" draft-buffer temp-buffer))

(defun revert-draft-source()
  (goto-char (point-min))
  (setq this-syntax (get-and-delete-line))
  (standard-revert-draft "source"))
```

See *E-L Emacs Utilities* for `get-and-delete-line`.

2.12 Examining error Derivatives

You will not need to read this section unless you have written a deriver that produces a derivative of class `error`. If you do, you may want to arrange for special display of such derivatives, which is controlled by the type of the artifact to which they are attached. As we shall see in section 3.5, error derivatives consist of a list of files (perhaps null) and a string (perhaps null as well). If both the list and the string are null, there is no error. The function to extend is:

o `examine-error-derivative-`*type artifact-cell descriptor file-list string*

- This function is called with an empty current buffer, which is to be filled with the display of an `error` derivative.
- In most cases, the *artifact-cell* argument is ignored. We will discuss one interesting use of this argument below.
- The *descriptor* is a string (see the first paragraph of chapter 3 for an explanation). The *descriptor* may say more about how to interpret the contents of the elements of *file-list* or how to interpret the *string*. It also says where files are (see the next item)
- The *file-list* is a list of strings. See below for how to use this argument.

  - The *string* is the part of the **error** derivative that does not describe the list of files.

There is only one reasonable use for the *file-list* argument: call the following function with an element of the *file-list* as the second argument (and the *descriptor* as the first argument).

- **insert-error-derivative-contents** *descriptor file*
  - Insert the file indicated by the arguments into the current buffer after point.

The reason for the *artifact-cell* argument is to support jumping (via **follow-link**, see the *E-L User's Manual*) from a buffer on the **error** buffer to a buffer on the artifact. To provide this kind of functionality, **examine-error-derivative-***type* sets up a buffer-local variable that holds the *artifact-cell*, and defines the variable **follow-link-hook** to point to a function which calls:

- **(examine-artifact (artifact-cell-pointer** *artifact-cell***))**

This causes the buffer on the artifact to be the current buffer, and it is customary after calling the above function to set point close to the error, based on information near point in the buffer on the **error** derivative. Later versions of this document will explain how to use parts of the artifact cell other than the pointer.

3 Derivative Class Extensions

To integrate a new tool into E-L, it is necessary to understand derivatives that the tool uses or produces. As discussed in the *E-L Users' Manual*, a derivative has a particular *kind*. Each kind belongs to a *derivative class*. In some cases, a derivative class can have only one kind, namely itself, e.g., `manuscript`. In other cases, the class is always a "stem" of the kind, in which case a "`%`" separates the class and a *species*, e.g., `object%sun3`—it does not make sense to have an object derivative without saying what its machine architecture is. In yet other other cases, a derivative class may itself be a kind, and may also serve as the stem of various kinds.

If you are writing a tool that produces only derivatives of existing derivative classes:

- Read only the first section below, to familiarize yourself with the general issues.
- Read the description of the class you will need (in whatever document describes that particular class)
- Proceed to the next chapter.

If you need to define a new derivative class, this entire chapter is relevant.

3.1 Equivalence of Derivatives

A fundamental premise of the artifacts system is that a derivative depends only on the contents of an artifact, the contents of its references, and so on, recursively. An equivalent formulation is that a derivative of an artifact depends only on its contents and on derivatives of references; this is closer to the way that derivatives are actually computed, as we shall see in the next chapter. The practical import of this constraint on derivatives is that if a derivative is deleted and rederived, the new derivative is the same as the old. This raises the question of what it means to be the "same". For example, a derivative of a given kind may be recorded as a file in a certain directory, and tools manipulating that kind of derivative will be given the file name as "the derivative". In such a situation, the meaning of "same" is surely not that the file name be identical—that would subvert the entire purpose of being able to delete the derivative and to rederive it. More likely, derivatives of this kind are the same if the contents of the files are the same, but this is only one instance of a definition of "same".

When we are being completely precise, we will use the term *derivative string*. As the name suggests, this is a short ASCII string, such as a file name or a file name and the decimal representation of an integer. A derivative string must not contain a newline character. We said earlier that a deriver produces a derivative, but what a deriver actually produces is a derivative string. To see this string, use a prefix argument (`C-u`) with `examine-derivative` (`C-z M-x`). For every derivative class, there must be a notion (but not an implementation) of equivalence for derivative strings for that class. The notion of a "derivative" is then abstract: it is a set of strings that are equivalent according to the equivalence relation for the derivative class in question. Then the rule that a deriver must produce the "same derivative" really means that any two derivative strings that it produces must be equivalent. There is a further rule that the functions that, informally speaking, take a derivative as an argument, but in reality take derivative strings as arguments, must behave the same on equivalent derivative strings. There are two ways in which derivatives are used:

- By functions that expose the derivative to a user, e.g., **`examine-derivative`**, **`export-derivative`**, as well as kind-specific commands, e.g. for executing or printing. In this case, what the user sees must depend only upon the equivalence class of the derivative.
- Uses of derivatives by other derivers. In this case, the requirement is that on equivalent inputs, the deriver produces equivalent outputs.

The second step is a recursive condition, reflecting the fact that derivatives are constructed recursively. The ultimate condition is the first condition—the user cannot tell the difference between equivalent derivative strings.

The reason for this seemingly technical discussion of the equivalence of derivatives is that on the one hand, the reproducibility of derivatives is central to the simplicity of the user's mental model of the artifacts system, and on the other, the ability to rapidly obtain derivatives incrementally (i.e., from derivatives of relatives) is central to the usability of the system. The fact that generic operations on derivatives like **`examine-derivative`** and **`export-derivative`** are tailorable by the extender of derivative classes allows wide variance between the user's view of the derivative and the data structure that tools see. To illustrate this point, we shall consider a derivative class *c* in which the identity of the derivative is that of a string of characters, perhaps quite long. Suppose the derivative is typically created as the contents of an artifact of type *t*, but where such an artifact has references, the *c* derivative of the reference is spliced in. This is not a toy example; the **`c-text`** derivative of **`c-source`** artifacts works in exactly this way. The naive implementation is simply to store the string of characters in a file, and to represent the derivatives by a *uid* (unique identifier), e.g., a generated name of a file in a directory dependent upon the kind. Then viewing of derivative with class *c* just puts the string of characters in a buffer, and export copies the string to a file; the above definition for equivalence as "having the same file contents" is correct.

There is a system-wide convention for *bad derivatives*—they are represented by a string whose first character is ASCII 2, i.e., `C-b`, for "bad". In documentation, a bad derivative is denoted, for example: **`^BParse errors`**. A deriver can generate a bad derivative as the derivative for any kind, as discussed in Section 4.5.1. The job server can also generate bad derivatives, as discussed in Section 4.5. A deriver never sees a bad derivative as input, and there is no need to define equivalence on them.

If a derivative is bad, then every derivative that depends on it will also be bad. Because of this cascading property, it is preferable for a deriver not to use bad derivatives, but to produce a good derivative if at all possible. The standard procedure for indicating problems with derivation by means of derivatives is discussed in detail in Section 3.5.

3.2 The `derivative-classes` Plex

The structure of the `derivative-classes` plex is very similar to that of the `types` plex (chapter 2); the only difference is that each line has an additional field that describes the representation of derivatives of that class. Just to make it official, a line in this plex has the form:

- *derivative-class resident-file auxiliary-file representation*

In a view on the plex, the last three fields are laid out vertically, and the representation field may extend over several lines. This latter field will be discussed in detail in section 3.3.

The commands to modify the `derivative-classes` plex are described only by their headers; for a full explanation, modulo the *representation* field, see the corresponding command for the `types` plex in Chapter 2.

- `new-derivative-class`
  derivative-class resident-file auxiliary-file representation `&optional` *initialization-file*
  - Obtaining the *representation* argument interactively sets up a recursive edit of the plex view, with the new entry for the derivative class and the cursor in the representation field. Highlighting sets off the editable region, which is terminated by a row of dashes. (Type `C-l` if the highlighting seems wrong.) Conclude the recursive edit with `C-M-c`, or abort the edit and the command with `C-M-]`.
- `repair-derivatives-class`
  derivative-class resident-file auxiliary-file representation `& optional` *repair-file*
  - Obtaining *representation* argument interactively sets up a recursive edit of its previous value. (The field is terminated by dashes only if it is the last derivative class in the buffer.) Terminate the edit as for the previous command.
- `delete-derivative-class` *derivative-class*

Also analogous to types are the following:

- `autoload-derivative-class` *symbol derivative-class string boolean*
- `require-derivative-class` *derivative-class*

There is now only one EMACS Lisp function family that needs extension by derivative-class:

- o `examine-derivative-`*derivative-class derivative-string*
  - This function is called only when the derivative is not totally bad, with the argument being the good part of the derivative.
  - Called with with the current buffer to be filled with a display of the given *derivative-string*. The only data in the buffer is the bad part of the derivative, if any.
  - `standard-examine-derivative` checks whether the derivative class structure is a simple file; if so, it inserts the file, and if not, it simply inserts the string. [This should be generalized to use the *representation* field of the plex to parse the *derivative-string*, and divide the buffer into sections showing all files, directories, and artifacts.]

3.3 Representation of Derivatives

In this section, we describe the syntax and semantics of the representation field of an entry in the `derivative-classes` plex. The rationale behind this field is to allow the extender to give a structural description of a derivative which will allow the artifacts system to take care of all storage management of derivatives. For example, consider the simple case in which a derivative string for a kind is simply a file name. In this case, the representation field will be the letter `f`, signifying file. As a string, the file name consists of a sequence of digits, interpreted as a file name relative to the canonical directory for derivatives of that kind, henceforth denoted *kind/*. Since there are many situations in which there is significant space savings if derivatives can share data, it is possible for the same file to appear in different derivatives. The question then becomes, when a derivative is deleted, should the file be deleted? Clearly, the answer is yes if and only if there are no more derivatives that refer to this file. To do the bookkeeping here, when a deriver reports a derivative of this kind for an artifact *p*, the system knows, because the representation field is `f`, that there is one more (maybe the first) reference to that file from a derivative, and places an *inhibition* on that file (i.e., an inhibition against deletion) on behalf of the *kind* derivative of *p*. When a derivative of an artifact is deleted, the system similarly knows to remove the inhibition of the file that was added by the derivative. When the last inhibition of a file disappears, the file itself is deleted. In short, the system does all the storage management bookkeeping; all the derivative class extender must say is "the representation of this derivative class is `f`".

A related use to which the representation field is put is "repository checking". Because the representation field describes how data is laid out, it is possible to periodically scan the repository, checking, for example, that all files that should exist do exist, and that all existing files are referenced by some derivative.

A secondary use of the representation field is in providing standard behavior for `examine-derivative`. In most cases, the representation field provides enough information to display a derivative-string in a way that depends only upon its equivalence class. For example, when the representation field is `f`, the display of it to a user is a buffer filled with the contents of the file. We could also use the representation to provide a standard behavior for equivalence of derivative strings, but as of now, there is no reason to actually implement such a function.

If all derivatives were represented as files, there would be no need for a representation field. However, the representation of derivatives can be quite complicated, in part because of the nature of existing tools, and in part because the design of derivers that are efficient in incremental situations often requires relatively complicated data structures for derivatives. The representation field can describe such data structures, and as a consequence, may itself be somewhat complicated. But as we have seen with the `f` example, if the derivatives are simple, so is the representation field that describes them.

The remainder of this section provides the specifics of what can currently appear in the representation field.[1] We will start with the simplest and most commonly used cases.

[1]It is our plan that when a new objectbase becomes available, the representation field will need to be extended to accommodate descriptions of derivatives that are stored in that object base. One might say that the only objectbase now supported is the Unix file system.

- **f**—a uid, relative to *kind/*. The uid names the file.
  - Syntactically, a uid appears as a string of digits.
  - If a new derivative refers to a file, an "inhibition" on that file is added (i.e., an inhibition against deletion).
  - When a derivative is deleted, its inhibition is deleted; when the last inhibition is deleted, the file is deleted.
  - [Currently, inhibitions are not created and removed as derivatives come and go. Instead, the derivative representation is used by a mark and sweep garbage collection algorithm that is run as part of the daily repository check.]
  - **standard-examine-derivative** displays the contents of the file named by the uid.

- `f{`*derivative-class*`}`—same as `f`, but the file is stored relative to the canonical directory for derivatives of the given *derivative-class.*

- **f%**—sames as **f**, but the derivative string has *species/uid* rather than just *uid.* The *species* does *not* include the %; in particular, */uid* indicates that the kind is exactly the derivative class.

- `f.`*extension.*—same as `f`, except that the name of the file in *kind/* has `.`*extension* appended to the uid. The *extension* does *not* appear in the derivative string. Extensions are provided only for compatibility with existing UNIX tools. They should not be used unless necessary.

- `f<`*representation*`>`—same as `f`, but in addition the file has contents structured according to the *representation*, recursively. (This is necessary if the file contains embedded references to data that must be protected against garbage collection.)

- `f{`*derivative-class*`}%.`*extension.*, `f{`*derivative-class*`}%`, `f{`*derivative-class*`}.`*extension.*, `f%.`*extension.*, plus all of these with <*representation*> tacked on to the end—the obvious combinations of the previous four.

- d—like `f`, but the entry in *kind/* is a directory, not a file.
  - Inhibitions are handled in the same way; when the last one goes, the contents of the directory are deleted, and then the directory itself.
  - **standard-examine-derivative** lists the file names in the directory, and provides for the user to be able to ask to see the contents of any requested file. (The assumption here is that equivalence of directories means that they have same-named files within them and that corresponding files have the same contents.)

There is no provision for directories with extensions, since no tool requires these.

- b{*type*}—like f, but the file is stored in the directory for bodies of a given artifact type. The file may also have inhibitions on behalf of the various artifacts whose bodies refer to it.

- b{*type*}.*extension*.—like b, but the file has the indicated extension.

- a—an artifact.

This concludes the description of the constructs that govern storage behavior of derivatives. The rest of the possibilities for a representation provide the ability to describe derivative strings with structure more complex than the above possibilities. First, there is the need to specify punctuation between pieces:

- '*character*—The *character* appears literally in the derivative string. A character is not quite a single ASCII character, but rather:
  - A single character other than \ stands for itself.
  - \digits stands for the character whose code is given by the *digits*, in octal.
  - \\\\ stands for the character \\.

- [*character* ...]—The *character set* construct, taken directly from EMACS regular expressions, except that \ obeys the above quotation convention.

- Except when preceded by a ' or inside a character set, all white space in a representation is ignored.

The representation `[\1-\377]` means that the derivative consists of a single non-null character (the possibility that the derivative consists of a newline is excluded by other means).

The remaining ways of constructing a representation all involve recursive use of representation. This includes at least the power of regular expressions. Precedence of the various constructs will be summarized at the end of the section.

- *representation representation*—concatenation. For example, the representation `f' f` says that the derivative string consists of a space-separated pair of files.

- *representation* | *representation*—alternatives. For example, `'Ff|'Dd` specifies that a derivative string is either F followed by the uid of file, or D followed by the uid of a directory. NB: Due to technical limitations in the regular expression machinery, if one alternative can match a substring of another alternative, the shorter alternative must appear last. In particular, an empty string is a legal alternative, but it must be the last alternative.

- *representation**—0 or more repetitions. For example, `[\41-\177]*` specifies a derivative string consisting of printing ASCII characters with no indirect storage.

- *representation**<*representation*>—0 or more repetitions of the first representation, separated by the second representation. For example, `f*<' >` specifies a possibly null space-separated list of files.

- *representation*+—1 or more repetitions.
- *representation*+<*representation*>—1 or more repetitions of the first representation, separated by the second representation.
- (*representation*)—grouping.

Finally, there are situations in which regular expressions are not adequate, because the derivative is tree-structured. For this, there is a way of introducing a nomenclature for a representation and then referring to a representation from within it.

- :*name*:*representation*—the name may consist of any characters other than : and =. There is no name hiding; the introduced *name* should be unique within the entire representation for a class.
- =*name*=—same form as *name* above. This may occur only within a *representation* named by :*name*:.

As an example, suppose the derivative is a list of elements each of which is either an integer or a parenthesized list of the same possibilities, recursively. Then the representation is:

- `:tree:([0-9]+|'(=tree=')')+<' >`

Strings matched by this representation: "`1 2 3`" and "`4 (5 6) 7`". It should be noted that this is only an example, and is not particularly useful, because it does not specify pointers that lead to other data. A more realistic example uses the <*representation*> option for the `f` syntax. Suppose that rather than parenthesized lists, the tree structuring is done by files. Because a file is syntactically the same as a sequence of digits, the syntax specifies that files are preceded with "`@`":

- `:tree:([0-9]+|'@f<=tree=>)+<' >`

[In the initial implementation, it is in fact a requirement that =*name*= be inside a representation associated with an `s` which is in turn inside the representation associated with :*name*:. This restriction guarantees that any particular string is parsable by a regular expression.]

The promised table of precedences, from highest to lowest:

* and +
juxtaposition
|
:*name*:

Precedence is of course overridden with ( ) and < >.

3.4 Fancy Techniques for Representing Derivatives

Recall the derivative class *c* from section 3.1. Suppose we notice that derivatives of class *c* tend to be large and that significant time is spent in re-deriving them. We might choose an alternate representation, for example:

- Either a derivative is a file,
- or else it is two files, one of which is a base file, and the other of which is a file of "deltas".

(The representation field would be "`f( f|)`".) Just to be specific, let us suppose that a file of deltas is a sequence of entries each having the format:

- *n*□*m*□*p*□*string*—after the *n*th character in the base file, replace the next *m* characters by *string*, which has *p* characters.

Assume such a file is sorted by *n*. It is a simple matter to create the string from a base file and a delta file, so that **`examine-derivative`** produces the same display as a naive representation. Similarly, it is possible to write a filter program to produce the stream of characters, which can then be fed to tools, like cc, that take the derivative as input. With not too much more difficulty, one can write a program to combine two deltas into a single delta, which may be useful in certain cases of incremental derivation. Note that deletion may need to change a delta representation to a direct representation.

Whether such a sophisticated implementation of derivatives would pay off is not really the issue. That has to be decided by measurements under typical patterns of use. The point is that there is sufficient flexibility in the system to allow such engineering to be done and yet hide it from the user.

A delta representation is one technique in managing derivatives. A second technique is "operator derivatives", where the idea is to package the information in a way that maximizes the amount of information that can be used incrementally. This can be done by treating the derivative as an "application" of an "operator" to some "operands". The operator comes from the contents of the artifacts and possibly some reasonably stable part of the derivatives of references. It has "formal parameters" in it. The operands come from the derivatives of the references. If the contents of the successor do not change, and if the parts of the references' derivatives used in forming the operator also do not change, a new artifact may share the operator part of its predecessor's derivative, even if the operands (the references' derivatives) have changed. This approach makes sense when references change more often than the contents of artifacts that reference them.

Operator representations of derivatives are useful when there is an existing Unix tool which follows embedded file names, and when the definition of derivative equivalence is intended to cater to this tool. (For example, the C preprocessor, when it encounters an **`#include`** macro, treats its argument as a file name.) The "formal parameters" are the file names, and the "application" is defined as running the tool in a context in which the file names are linked to appropriate parts of the derivative. In this case, a derivative may be represented as follows:

- $f\Box u_1\Box\cdots\Box u_k$
  - f is a uid relative to the directory *kind/*.
  - Each u_i is a name by which a file is referenced from within f by the Unix tool. (In the C preprocessor example, the derivative would contain `#include "`u_i`".`) It must be guaranteed that the u_i associated with different files f are distinct.

It might seem that, because the u_i are nearly arbitrary choices, in some sense the derivative is not determined entirely by the artifact, but of course, we have a subtle definition of derivative equivalence in mind, specifically, let $f\Box u_1\Box\cdots\Box u_k$ and $f'\Box u'_1\Box\cdots\Box u'_{k'}$ be two strings representing derivatives. They are equivalent when $k = k'$ and when they pass the following test:

- Take any directory d in which the names u_i and u'_i are not used, and take any absolute path names $f_1, \ldots, f_k$. Link d/f_0 to *kind*$/f$; link d/u_i to f_i, $i = 1, \ldots, k$; and run the Unix tool in directory d. Alternatively, link d/f_0 to *kind*$/f'$ and d/u'_i to f_i, and run the tool. There will be no difference in the effects of the two tool invocations.

Loosely speaking, f and f' have the same contents, except that where f refers to a file u_i, f' refers to u'_i. It turns out that it is not necessary to literally implement this definition of equivalence; the main point here has been to give an example of derivative equivalence which is tailored to the semantics of an existing tool. It is also necessary to define viewing and exporting of the derivative appropriately; it would not be kosher to expose the u_i or u'_i. How to do this is a second-order detail which we do not discuss further here.

A downside to cleverly engineered representations of derivatives is that they become more complex to use, and that all the tools that use the derivatives of a particular class must deal with that complexity. However, our experience has shown that derivatives of a given class are ultimately consumed by only one tool. For example, `c-source` derivatives may be spliced into other `c-source` derivatives, but are eventually processed to produce `c-file` and `h-file` derivatives.[2]

3.5 `error` Derivatives

Many derivers detect errors in their inputs, for example, a compiler will detect syntax errors. Almost all such errors should be reported as derivatives with class `error`, for reasons discussed in the last paragraph of section 3.1. As we noted in section 2.12, such derivatives logically consist of a list of files and a string. The encodement into a derivative string is as follows:

- Zero or more uids relative to the kind in question, separated by spaces.
- Either nothing else, or a vertical bar (`|`) followed by arbitrary text (not containing a null, `^A`, or newline).

A deriver produces such a derivative in the same way that it would produce any other derivative.

[2]A current exception to this is `manuscript` derivatives, which are used not only in typesetting but also in extracting program text. This is a temporary aberration resulting from historical use of files.

3.6 universal Artifacts

The artifact type `universal` allows the specification of *any* desired derivative (well, almost). The contents of such an artifact is a series of lines having one of two forms:

- comment—empty or beginning with a non-alphabetic.
- *kind*$\boxed{\text{t}}$*derivative-string*

The universality comes from the fact that a `universal` artifact has a derivative of kind *k* if and only if *k* is among the kinds listed in its contents, and the derivative is given by a derivative string associated with *k*, with suitable provisions for files, directories, and artifacts, discussed below.

First, a caveat. To construct a `universal` artifact, you must understand the precise form expected of the derivative string. The system will help you as far as it can—it will use the representation field of the `derivative-classes` plex to ensure that each derivative string has the right format—but there may be subtle properties that are required of a derivative of a given kind that are not grammatically specifiable: `universal` artifacts are for professionals; do not attempt to use them at home.

Now we describe the provisions for the parts of a derivative string that "point" to data. The simplest of these is an artifact pointer, indicated in a derivative representation by `a`. In a `universal` draft, simply insert an artifact reference at the point in the derivative string where you want the artifact pointer. In the draft, or in the `universal` artifact obtained by committing, this will look like an ordinary artifact reference. But in the derivative string, it will appear as the desired artifact pointer.

A derivative representation indicates a file by `f`[`{`*derivative-class*`}`][`%`][`.`*extension*`.`] or by `b{`*type*`}`[`.`*extension*`.`]. Of these possibilities, only `f`[`.`*extension*`.`] can be produced by the deriver for `universal` artifacts. There are two ways of specifying the derivative string for this case, allowing you to choose between data integrity and reduced disk space. The reason for allowing you to sacrifice data integrity is that there are files outside the artifacts system that one cannot afford to copy into the artifacts system, for example, compiler executables or a Lisp world. (Not to mention the fact that some compilers must reside at a particular place in the file system.) While it is a practical necessity to allow pointers out to the mutable world, it is also desirable to retain the immutability properties of artifacts and derivatives. One way to compromise between these irresoluble differences is to specify a file in a `universal` draft by an absolute path (i.e., beginning with a `/`), that you highlight (use `C-z C-h`) to distinguish it from other text in the derivative string. Highlighting this text first checks that the file exists, and does not succeed if there is no file. If the highlight is successful, you will see that the path is now followed by two numbers, the length in bytes and a check sum, together called a *certificate*. (Computing the check sum may take a while on large files.)

One of the uses of a certificate is in derivation. We now explain the "suitable provision" for a file reference that is an absolute path. The deriver checks to see that the indicated file still exists, and if so, whether it still has the same certificate. If so, it allocates a new uid for the kind in question, and establishes a soft link from this uid (possibly with an extension) to the absolute path. If not, it produces a bad derivative.

Because the derivative "file" is a link, it is subject to deletion or change not under control of the artifacts system; this is the price paid for not duplicating the data. However, to provide a modicum of integrity, the repository checker periodically runs the following algorithm:

- For every `universal` artifact:
  - For every absolute path in a derivative string:
    - If the derivative exists:
      - Let the uid for the file be u.
      - If the file u (i.e., the file to which it is linked) has a certificate different from that indicated in the argument, post a `notice` to the creator of every artifact in the transitive closure of references to the artifact in question, unless there is already such a notice.

The system does *not* delete any suspect derivatives, because the file in question may have been only temporarily corrupted. However, while the system does not pile up redundant notices for any user, the notice will keep coming back until the problem is fixed. Thus, even though the integrity of the derivatives is not guaranteed, violations of that integrity will, with high probability, be reported to an interested party within a finite amount of time, and will effectively persist until the problem is cleared.

Another way to specify a file part of a derivative string is to use an "internal pointer". In this case, the integrity of derivatives is guaranteed, and the file is stored within the artifacts system. To use this, highlight the character `@`, which will turn into `@`*n* for some *n*, and a divider of the form `----File` *n* `---` $\cdots$ in inverse video) will appear at the end of the buffer. Insert the desired contents after the divider. If there are several internal pointers to files, there will be several dividers, effectively dividing the buffer up into "internal files". When the draft is committed, the data for each internal file will be stored in a real file, and derivation will establish links to these files, much the same as for an absolute path, except that because the file is under control of the artifacts system, there is no danger of its becoming invalid. [It may be necessary to provide a mechanism for giving execute permission to real files.] If you edit an artifact, you can then edit the internal files. The new artifact will share files with a predecessor if possible, but will create new files as necessary.

Similar provisions could be provided for directories, represented by `d` in a derivative representation. However, the need for these is judged to be low priority, and the details of the user documentation, not to mention the implementation, will be postponed until there is a felt need.

4 Derivers

The subject here is how to write derivers and install them. This chapter assumes that the reader has used the E-L system, and so is familiar with jobs being run on the user's behalf. It also assumes that the reader already understands the representation of artifacts whose type is that of interest and the details of their representation for the desired derivative class.

4.1 How Jobs are Scheduled

The purpose of this section is to provide an extender with an accurate mental model of how jobs are run, and thereby provide a framework in which the specifics of remaining sections make sense.

Scheduling jobs is a distributed process: there is no central scheduling agent. An artifact typically has some jobs scheduled for it when it is committed, and the user can explicitly schedule jobs using `derive-from`. But requirements for additional derivatives also arise during derivation, and these requirements may entail further scheduling. If so, exactly the same algorithm and the same shared data structures are used as in the case of scheduling, either implicitly or explicitly, that originates in an editor session.

The scheduling problem is this: given an artifact p of type t, and a kind of k for a desired derivative of p, how does the system ensure that the derivative will eventually be computed? There are several plexes that are involved in scheduling jobs. To simplify the description of how jobs are scheduled, this section treats all of them as conceptually being relations, and ignores the parts of the plexes not having to do with scheduling.

- `present`—each tuple in this relation corresponds to a *job server*, i.e., a process capable of running jobs. The fields are:
  - *id*—unique to a job server; it can be used to communicate with it.
  - *job-server-class*—a string that says what kinds of jobs this job server is capable of performing.
  - *idle*—a boolean, indicating whether this job server can be sent a new job.
  - *current-jobs*—a set of pairs of the form $\langle p_i, K_i \rangle$, where p_i is an artifact pointer and K_i is a set of kinds, indicating which derivatives of which artifacts are presently being computed by this job server.
- `future`—each tuple in this relation is a job that is waiting to be run. The fields are:
  - *artifact*
  - *kind*
  - *priority*—low numbers win.
  - *timestamp*—when the entry was made; used in tie-breaking in case of identical priorities.

Important invariant: if $\langle p, k, \cdots\rangle$ is in the **future** plex, then there is no $\langle p, K\rangle$ with $k \in K$ in a *current-jobs* field of any entry in the **present** plex.

- **workers**—this relation has the fields:
  - *worker*—a string with an associated executable.
  - *tool*—the name by which the outside world refers to a function within the executable. (For derivers, the name of the tool is generally of the form **derive-*type*/*kind***. We will see later that there are tools other than derivers.)
  - *job-server-class*—says which job servers (elements of the **present** plex) are allowed to run this tool.

  A given worker may appear in many tuples.

- **derivers**—fields:
  - *tool*—the name of the tool capable of computing a set of derivatives from an artifact, or else deciding that such a derivative definitely does not exist.
  - *type*—the type of some artifact.
  - *kind-set*—the set of kinds that will be produced by the deriver.

  The **derivers** plex does *not* say that the derivative of a certain kind necessarily exists; it may be necessary for the tool to take a closer look to decide that.

  The reader is warned that "conceptual" is the operative word for this relation—if represented literally as a set of tuples, it would be infinite; moreover the *kind-set* in each tuple might be infinite. The finesse will be explained in section 4.4.

We now answer the scheduling problem quite succinctly:

- Given an artifact p and kind k, is there some $\langle p, K\rangle$ with $k \in K$ in the *current-jobs* field of some entry in the **present** plex? If so, the job is already being run. (In practice, the job server can be told to notify the new requester when the job is done.)

- If the job is not under way, let t be the type of p, and ask whether there exists a tool T, a kind set K, a worker w, a job-server-class c, and a job server id j such that:
  - $\langle T, t, K\rangle \in$ **derivers** and $k \in K$
  - $\langle w, T, c\rangle \in$ **workers**
  - $\langle j, c, \text{true}, \cdots\rangle \in$ **present** (i.e. an idle job server)

  If so, send j a message telling it to get busy deriving kind k from artifact p using tool T in worker w. Moreover, add $\langle p, K\rangle$ to the *current-jobs* field of j.

- If there do not exist T, K, w, and c such that the first two conditions hold, then the job is not schedulable.

- Otherwise, put p and k into the `future` plex (with suitable priority and timestamp).

There is one other situation in which scheduling is done, namely when a job server is about to become idle. Before setting its *idle* field to true, a job server with class c asks if there exists an artifact p of type t, kind k, kind-set K, tool T, and worker w such that:

- $\langle p, k, \cdots \rangle \in$ `future`
- $\langle T, t, K \rangle \in$ `derivers` and $k \in K$
- $\langle w, T, c \rangle \in$ `workers`

If so, it picks the p, k with the smallest priority/time in the `future` plex, and removes it along with any other p, k' where $k' \in K$. This maintains the invariant relating the `present` and `future` plexes. Only if no such p, k, T, K, w exists does the job server become idle.

As stated earlier, a *tool* in the above relations must be connected with a function inside a worker. However, a *worker* and a *job-server-class* have no meaning other than their use as strings in making connections in these relations. In other words, a worker can be renamed at will in the `workers` plex, without effect on scheduling. (It can't be changed to an existing *worker* though). Similarly, changing a *job-server-class* uniformly throughout the `workers` and `present` plexes to a previously unused job server class will not affect scheduling subsequent jobs.

Observe that the `present` and `future` plexes are modified as jobs are scheduled, but the `workers` and `derivers` plexes are not. Rather, they are modified by the extender to direct jobs to the right tool in the right worker for the right job server. Section 4.3 and 4.4 discuss manipulating these plexes, and the subsequent sections describe how to write a deriver. Observe that the order in which you actually do an extension is the reverse of the order of the presentation—first you have to write the derivers, then put the tool in the `derivers` plex, and then install the worker.

4.2 The hosts plex

(A slight detour before details of deriver installation). The `hosts` plex is not used during scheduling, but it is relevant here, because it is used by job servers. A line in the plex:

- *host machine-type default-job-server-class-variant*
  - The *machine-type* is not at the discretion of the extender, but is computed from the *host* and cached in the plex. See *Porting E-L* for a full discussion.
  - The usual start-up of a job-server on the *host*, e.g., with `M-x start-job-server`, concatenates the *machine-type* and *default-job-server-class-variant* to obtain the job-server-class for the job-server's entry in the `present` plex.

It is also possible to start a job server with a given value for its job-server-class, in which case it ignores the `hosts` plex. [PRESENT REALITY: there's no job-server-class apart from the *machine-type* of the host.]

The `hosts` plex is manipulated with the following commands:

- **new-host** *host default-job-server-class-variant*
  - The *host* must not already be in the plex.
  - If the *host* is not the one on which the command is issued, this command will do a remote-shell to the *host* in order to compute the machine-type. Thus, you must have remote login privileges on that machine.
  - A common value for the *default-job-server-class-variant* is the null string.
- **repair-host** *host default-job-server-class-variant*
  - Used to change the variant.
- **delete-host** *host*

4.3 The workers Plex

In the previous section, we described the **workers** plex as conceptually a relation. Here we describe its actual representation, and the connection to the conceptual relation. A line in the **workers** plex has the format:

- *worker executable-file machine-type* $\{tool_i\}_i$
  - The *machine-type* indicates on which architectures the worker can run. While this field can contain the empty string, for now it oughtn't. [1]
  - Let a line be w, f, m, $\{T_i\}_i$. Its contribution to the conceptual relation of section 4.1 is the set of tuples $\langle w, T_i, c_j \rangle$, for all i and j.[2]
  - The *executable-file* field is the file outside the repository from which the installed executable for the worker was copied or moved. In a system newly installed from a distribution, this field will refer to the file in the distribution. Ultimately, it will be possible to install a worker via an artifact, in which case this field will be have the artifact and a machine variant pair *m-v*, where the executable is the **executable%***m-v* derivative of the artifact.

In a view on the **workers** plex, the fields are displayed vertically, with the tools underneath the executable. [PRESENT REALITY: just one line per worker entry.]

[1]PROPOSED: If this field is null, it works for all machine types (and is presumably a shell-script).
[2]There is presently only one job server class, which is currently implicit in each line of the **workers** plex.

```
c-worker-sun3 sun3 C /usr/local/e-1-distribution/sun3/c-worker-sun3
                     derive-c-source/c-text
                     derive-c-module/c-and-h-file
                     derive-c-module/object-sun3
                     derive-c-module/executable-sun3

c-worker-mips mips C /usr/local/e-1-distribution/mips/c-worker-mips
                     derive-c-source/c-text
                     derive-c-module/c-and-h-file
                     derive-c-module/object-mips
                     derive-c-module/executable-mips
```

The `workers` plex is manipulated with the following commands:[3]

- `new-worker` *worker machine-type job-server-class-list file-or-artifact tool-list*
  - The *worker* must not already exist, and all the tools in the *tool-list* must already exist (e.g., be registered in the `derivers` plex).
  - When called interactively without a prefix argument, this will prompt for an artifact argument for the *file-or-artifact*; otherwise, it will prompt for a file. With one `C-u`, the file will be copied; with two, it will be moved.
- `repair-worker` *worker machine-type job-server-class-list file-or-artifact tool-list*
  - The arguments are the same as for `new-worker`.
  - This command first ensures that no job-server is actively running the worker.
  - Default values for the arguments are supplied using the values already in the plex. You must confirm that you really do want a new executable installed.
- `delete-worker` *worker*
  - This command first ensures that no job-server is actively running the worker.

As an aside, we note that a tool may be present in a worker executable, but be prevented from running on certain hosts, by making no entry in the `workers` plex for a given job server class. So to prevent cc compilation from running on a certain host, say, one could give it a default job server class `X` (in the `hosts` plex), and set up the `workers` plex as follows:

```
c-worker-sun3 sun3 X /usr/local/e-1-distribution/sun3/c-worker-sun3
                     derive-c-source/c-text
                     derive-c-module/c-and-h-file
```

Thus, two job servers, one with class `C` and one with class `X`, can share executables, but only the job server class `C` will receive jobs where the tool is `derive-c-module/object%sun3` or `derive-c-module/executable%sun3`.

[3][PRESENT REALITY: a *tool-list* has only one *tool*, but there are extra commands `repair-worker-new-tool`, `repair-worker-delete-tool`, and `repair-worker-executable`.]

4.4 The derivers Plex

Recall from section 4.1 that the **derivers** plex is conceptually an infinite set of tuples of the form $\langle T, t, K\rangle$. The finesse to gain finiteness is a two-fold reliance on regular expressions. First, there is the fact that a regular expression itself can represent an infinite set of strings, namely those that it matches. Second, there is the fact that using the `\(...\)` construct of Gnu EMACS (see the manual), a substring of a matched string can be extracted, and using the `\i` construct (ibid.) for replacing the i^{th} match, the extracted string can be substituted into *another* regular expression. This double reliance on regular expressions disposes of two levels of infiniteness—the infinite relation with infinite sets inside. An example will follow the general description of a line in the **derivers** plex:

- *artifact-type tool kind-re kind-re-replacement*
  - The semantics of this line is described by giving the tuples that it contributes to the conceptual relation used in section 4.1.
  - Let T be the tool, t be the artifact type, *re* be the *kind-re* and *re-r* be the *kind-re-replacement.* For every string s matched by *re*, there is one tuple contributed to the conceptual relation. To obtain it, take the substrings of s located by *re*, and substitute these for occurrences of \i in *re-r*, obtaining a new regular expression *re′*. The tuple is $\langle T, t, K\rangle$, where K is the set of all strings matching *re′*.
  - The use of *kind-re-replacement* is almost redundant. It is currently used to allow one to specify grouping in *re′* without having it misinterpreted in *re* as a substitution position.
  - *kind-re-replacement* should always be written so that the final *re′* matches a *sub-language* of *re*. That is, no deriver should produce a derivative that cannot be directly scheduled.

To get a feel for how this works in practice, we look at two real examples. First consider derivers that obtain object files from `c-module` artifacts. (See the *C and Unix Programming Manual* if you are not familiar with this artifact type.) It is necessary to derive the object files for different machines with different tools, so we use the *kind-re* to distinguish them. In the example entry below, the derivative class is `object` or `error`; the variant is covered by the regular expression `\(-[A-Za-z0-9]+\|\)`

```
derive-object-sun3 c-module \(object\|error\)%sun3\(-[A-Za-z0-9]+\|\)
                            \(object\|error\)%sun3\2
```

For the strings s = `object%sun3`v or `error%sun3`v, the tuple is:

⟨`derive-object-sun3`, `c-module`, {`object%sun3`v, `error%sun3`v}⟩

Here v ranges over the empty-string and strings beginning with - followed by alphanumeric. Thus, there are an infinite number of tuples, but the kind set in each is finite, namely, it has two elements. This reflects the fact that producing an object file is inseparable from producing the error file associated with the compilation.

For the second example, consider **shellscript** artifacts (ibid). Here, there are two new issues: the **manuscript** derivative is obtained by a type setting tool, but all other derivatives are obtained, if they can be obtained at all, by a tool that runs the shellscript. So the relevant regular expressions are:

```
derive-shellscript-manuscript  shellscript  manuscript   manuscript
derive-shellscript-others      shellscript  \~manuscript \~manuscript
```

The `\~` operator means the complement of the regular expression following it. The second line here contributes the tuple:

$$\langle \texttt{derive-shellscript-others}, \texttt{shellscript}, \{\text{all strings} \neq \texttt{"manuscript"}\}\rangle$$

So here there is only one tuple, but its kinds-set is infinite.

The following commands modify the **derivers** plex.

- **new-deriver** *tool artifact-type kind-re kind-re-replacement*
  - The *tool, artifact-type* pair may not already exist.
  - The *artifact-type* must already be registered in the **types** plex.
- **transpose-derivers** *mark point*
  - This command can be issued only from a view on the **derivers** plex, with a mark set on line and the cursor on another. It exchanges the two lines.
  - Rationale: the order of lines in the **derivers** plex is significant, so it must be possible to change the order of entries.
- **repair-deriver** *tool artifact-type kind-re kind-re-replacement*
  - The current values of the last three arguments are presented as defaults, making it easy to change only one of them.
- **delete-deriver** *tool*

4.5 Writing a Deriver

This section gives a language-independent description of how to write a deriver. We will discuss separately how to map this abstract specification to the concrete rules for different implementation languages, currently C, Emacs Lisp, and Common Lisp.

A deriver is implemented as a set of functions $\{f_i\}_i$, many of which can be null, indicating some default behavior. These functions are called by the general derivation mechanism, which also has as arguments the artifact under derivation, which we shall consistently refer to as p, and the set of kinds, each of which was matched by the *kind-re* field for the derivers's line in the **derivers** plex. A deriver is given the chance to consider *relatives* of p, i.e., artifacts that are historically related to p. For now, the only relatives are immediate predecessors, but this may be generalized in the future.

The functions f_1, f_2, and f_3 take as their first three arguments:

- The *body* of an artifact (either *p* or one of its relatives).
- The artifact *cell.* The deriver should not modify any slots in the cell.
- The *location* of an arbitrary datum, e.g. the location of the contents of the artifact; f_1, f_2, or f_3 can store in this location any data that must be passed between f_i's. *location* initially contains a null, and f_1, f_2, or f_3 can set and read its contents. f_4 and f_5 can read *location*'s contents, because the value in *location* becomes the value of the *contents* slot of the corresponding dossier (see below).

The remaining arguments and f_4 and f_5 are described below.

To declare a deriver, enter a call on the following function in the initialization of an appropriate, necessarily language-specific, worker:

- **`enter-deriver`** *name* $f_1, \ldots, f_k$ *job-server-incremental*
  - The *name* is a string that names the tool. The standard convention is to use *type/kind*, and when more than one kind can be produced, to choose the principal one or a suggestive term. The name is not parsed by the system; as explained in section 4.1, it is used only as a match of the same name in the `derivers` plex.
  - The f_i must occur in the order listed below.
  - *job-server-incremental* should be true iff none of the derivatives produced by the deriver ever embeds the artifact pointer of the argument artifact (or the pointers of references if these do *not* appear in the derivative strings of the references). Otherwise *job-server-incremental* should be false. More precisely, *job-server-incremental* is true when the job server can apply its default incrementality machinery based on body strings and the derivative strings of requisite and reference artifacts.

Now the specifics of the f_i. Each one is given a suggestive name, and an argument/result specification.

- o f_1 = *computable-kinds*: *body type location requested-kind-sequence* $\longrightarrow$ *kind/uid-request-sequence*
  - *requested-kind-sequence* is the set of requested kinds.
  - The result is a sequence of kind/uid-request objects composed of:
    - * A *kind* for *p* that this deriver will produce given *requested-kind-sequence.* This may be a kind not in *requested-kind-sequence.* For example, a deriver that is asked to produce an object file might yield both an object and an error derivative.
    - * *uid-count*, an integer that indicates the number of uids that likely to be needed for *kind.* This is an estimate needed for efficiency only. Calls to functions that return uids (see 4.6) will always succeed, but requests beyond the number given by *uid-count* will result in additional (and somewhat costly) communication with the job server.

  - The default result (i.e., the one used if a function is not supplied), is $\langle\langle k, 0\rangle\rangle_{k \in \textit{requested-kind-sequence}}$, i.e. the kinds of *requested-kind-sequence* with no uids requested.

- f_2 = *requisite-kinds*: *body type location* $\rightarrow$ *kind-sequence*
  - The result is the set of derivative kinds for p that are needed for this deriver (and presumably computed by some other deriver).
  - For example, in deriving an object file, the result would be {`c-file`}.
  - The default result is the empty set.

- f_3 = *reference-handle-kinds*: *body type location integer* $\rightarrow$ *handle-kind-sequence*
  - The result is a set of handle/kind pairs that specify which kinds of derivative are needed from which references. The same handle may appear more than once, with different kinds.
  - This function is not called if there are no references.
  - The *integer* is the number of references of the artifact, a convenience for situations in which the same kind of derivative is needed from each.
  - For example, to compute the `c-text` derivative of `c-source` artifacts, the result is $\{\langle 0, \texttt{c-text}\rangle, \ldots, \langle n-1, \texttt{c-text}\rangle\}$, where n is the *integer* argument.
  - The default result is the null set.

The next two functions use a data structure called a *dossier*, summarizing information about derivatives of an artifact and its references. The fields of a *dossier* are:

- *body*—The deriver should not modify this field.
- *contents*—What was placed in *location* by f_1, f_2, or f_3.
- *kind-derivative-sequence*—A list of kind/derivative-string pairs, which we denote

  $$\{\langle \textit{requisite-}k_i, \textit{requisite-}d_i\rangle\}_i.$$

  The *requisite-k_i* are from the result of *requisite-kinds* on the artifact; the *requisite-d_i* have been obtained and filled in by the job server. For p, all of the *requisite-d_i* will be non-null, because they will have been computed if necessary. For relatives, any or all of the *requisite-d_i* may be null, because they are supplied only if they already exist.

  The deriver should not modify this field.

- *handle-kind-derivative-sequence*—A list of handle/kind/derivative-string triples, which we will denote by $\{\langle h_j, k_j, d_j\rangle\}_j$. The h_j, k_j are from the result of *reference-handle-kinds* on the artifact and are in the same order; the d_j have been supplied by the job server. As in the previous case, the d_j for p will all be present, while for the relative, only the previously existing derivative strings will be non-null.

  The deriver should not modify this field.

There are three more f_i. The first is responsible for comparing relatives with p, the second, puts all the pieces together, and the last is used when a deriver is likely not to terminate.

- f_4 = ***compare-relative***: ***dossier dossier kind-derivative-sequence*** $\longrightarrow$ ***boolean*** $\uplus$ ***hint***
  - The first argument is for p; the second, for a relative, call it p'.
  - Let $\langle\langle k, \textit{computable-d}_k\rangle\rangle_{\langle k,x\rangle \in \textit{kind/uid-request-sequence}}$ denote the value of the *kind-derivative-sequence* argument (where *kind/uid-request-sequence* is the result of f_1). Each *computable-d_k* is the derivative string of p' for kind k (supplied by the job server). These derivative strings might be null (they are not computed, only looked up), but not all of them are null, or else this function would not be called on this p'.
  - A result of true means that output derivative strings of p' are valid derivative strings of p. A result of false means not only that the derivative strings of p' are not valid, but also that there is nothing in the extant derivative strings of p' that is of use in the derivatives of p. A result that is a *hint* has some useful information for derivation. It will be passed along to the next function. A *hint* is an arbitrary data structure of the deriver writer's choosing. It must incorporate everything that can be gleaned from p', because p' itself will not be seen again.
  - This function is called only when either the bodies are different (not necessarily the contents) or the derivative strings of references are different (as strings, not necessarily inequivalent in the sense of section 3.1); otherwise, the job server knows that the answer should be true.
  - The default result is false (useful when a relative's derivatives are valid only when the relative's body and derivatives are identical, and are otherwise useless).
- f_5= ***derive***: ***dossier kind-derivative-set-sequence hint-set*** $\longrightarrow$ ***derived-results***
  - The *dossier* is for p.
  - Denote the *kind-derivative-set-sequence* argument by $\{\langle \textit{derive-}k_m, D_m\rangle\}_m$. Each *derive-$k_m$* is a k such that $\langle k, x\rangle \in$ *kind/uid-request-sequence* and for which no derivative string was available from relatives. Each D_m is a (possibly empty) set of derivative strings of kind *derive-k_m* of relatives of p. If possible, an element of D_m should be used in the result. You may think of this as a universal deriver-supplied hint (as opposed to a *hint*, which is extender-provided).
  - The *hint-set* is constructed from calls on the previous function. (In languages that represent sets as ordered sequences or lists, the elements are in no particular order.)
  - *derived-results* is $\{\textit{derive-}d_m\}_m$, This is a sequence of derivative strings parallel to *kind-derivative-set-sequence.*
  - Deriver requests for uids and filenames, streams or standard text handles associated with them should be made through the functions described in 4.6.

  - This function *must* be supplied; it has no default alternative. However, it is not called if all derivatives can be computed incrementally.

- f_6 = ***abort-job-hook: abort-job-data kind-sequence*** $\rightarrow$
  - The deriver writer should supply this function exactly when the deriver cannot guarantee termination, but needs to return an `error` derivative (e.g. derivers that run LaTeX or COMMON LISP compilers).
  - This function executes asynchronously during *compare-relative* (f_4) or *derive* (f_5), when a user executes a `M-x abort-job` command on the derivation.
  - ***abort-job-data*** is a generic, derivation-local, pointer maintained by the worker kernel. It is set by calling

    `set-abort-job-data:` *generic-pointer* $\rightarrow$

    which copies *generic-pointer* to *abort-job-data*. (An error is signaled if `set-abort-job-data` is called when no f_6 is supplied, or in any function other than *compare-relative* or *derive*.)

    abort-job-data can be read with

    `get-abort-job-data:` $\rightarrow$ *generic-pointer*
  - ***kind-sequence*** is $\langle k \rangle_{(k,x)\in kind/uid\text{-}request\text{-}sequence}$, where *kind/uid-request-sequence* is as described in f_1.
  - The only way that f_6 can affect the derivation is to either
    1. use `set-abort-job-data` and return normally, or
    2. use any language-specific non-local return.
  - If f_6 is not supplied or if `set-abort-job-data` has not been called, then `M-x abort-job` causes the derivation to return aborted derivatives:

    `^BJob aborted by user.`

4.5.1 Reporting Errors

There are three independent means by which a deriver might indicate a problem with derivation:

- It might return a bad derivative.
- It might return a non-empty derivative string of class `error`.
- It might call some worker-kernel-supplied error function. This method should be reserved for reporting errors that cannot be caused be erroneous user input (in other words, for errors that "should not happen").

Many errors are handled by the job server, and require no action by the deriver. The job server handles errors due to

- non-existence of a suitable deriver,
- catastrophic worker failure (see below for implementation-language-specific details),
- a bad requisite or reference derivative, and
- requested kinds that are not among the computable kinds.

Whenever a deriver returns an **`error`** derivative or a bad derivative, the job server posts a notice to the job's requesters. The type of the notice is **`derive error`**. Displaying a notice with this type will invoke **`examine-problems`** on the artifact and relevant derivatives.

The deriver must report all other errors, e.g. parse errors in a program-text artifact. The sequel concerns details and strategies for reporting problems or errors to the E-L users using one of these methods.

Bad Derivatives As defined in Section 3, a bad derivative is one whose derivative string begins with `C-b`. A completely bad derivative string has only one `C-b`, namely the first. The rest of the string is completely uninterpreted by the system but appears in automatically issued notices.

A *partially bad derivative* is a bad derivative of the form

`^B`*message*`^B`*usual-derivative*

The system does not interpret *message*, but uses it for displaying notices to the user (see below). However, *usual-derivative* must conform to the format for this derivative class as registered in the **`derivative-classes`** plex. The system recognizes the contents of *usual-derivative* for the purposes of garbage collection just as it does an ordinary good derivative.

Error Derivatives A deriver may return a non-empty derivative string of class **`error`** provided that its ability to do so has been registered in the **`derivative-classes`** plex.

This is the most flexible method of reporting an error, since any information can be stored in the contents of the **`error`** derivative. Many kinds of user support can be provided in the function that displays the **`error`** derivative for a particular artifact type. Consult the **`derivative-classes`** plex for the allowable format of **`error`** derivatives.

Dispatch for display of non-empty **`error`** derivatives is[4] by artifact type alone.

There is a subtle point to be observed about **`error`** class derivatives. A good **`error`** derivative must, like all other derivatives, functionally depend on the contents of the artifact from which it is derived, or on the derivatives of the artifact's references. Thus, if the error being reported is due to a system resource limitation, or is otherwise *not* dependent on the contents of any artifact, the error derivative should be partially bad. The user is then properly alerted that deleting the derivative may destroy unrecoverable information. Furthermore, a partially bad **`error`** derivative string will not be re-used as a derivative for successor artifacts, nor will it be used by related derivatives.

[4]In the near term, at least.

It is up to the deriver writer to decide what combination of bad, partially bad and `error` derivatives to return when a derivation problem arises. Returning a partially or completely bad derivative is suitable only when it is acceptable to abort jobs for which the job reporting the error is a prerequisite.

4.6 Language-Independent Utilities

The following utilities are provided in both C and LISP worker kernels.

- **`uniform-handle-kinds`** *first last kind* → *handle-kind-sequence*
  - Return a sequence of handle-kind pairs of the form $\{\langle i, kind\rangle\}_{i=first,\ldots,last}$.
  - Signal an error if *first* < 0.
  - Return an initialized, empty sequence if *last* < *first*.
  - In practice, often called by a *reference-handle-kinds* function.
- **`derivative-path`** *kind filename* → *path*
  - * Return a fully-qualified path name for a file holding derived information for *kind* and having the given *filename*.
- **`derivative-area`** *kind* → *path*
  - * Return the full path name for the directory holding derived information files for the given `kind`.
- **`stream-and-uid`** *flag area extn* → *stream uid*
  - Return a *stream* open for writing in a directory determined by *flag* and *area*.
  - If *flag* is 0 then *area* is interpreted as a derivative kind and the file associated with *stream* is in the directory for that kind. If *flag* is 1, *area* is interpreted as an artifact type name, and the associated file is in the directory for that type.
  - The file associated with the returned values is guaranteed to be previously unused in the associated directory.
  - *uid* is the file name without extension or leading path.
  - If *extn* is *null* then the name of the associated file is *uid*, a string of digits.
  - For all other values of `extn` the associated filename is *uid* followed by a "." and then the value of *extn*.
- **`filename-and-uid`** *flag area extn* → *filename uid*
  - Like **`stream-and-uid`** except that the first value returned is the name of the associated file.
  - *filename* is *not* a fully-qualified path name, but (can be used as) the name of a file in the associated directory.

- Use this function only if you want the filename for purposes other than simply opening a stream on it. It is more efficient to use `stream-and-uid` than to open a stream on a file name generated by `filename-and-uid`.

  The functions `stream_and_uid` and `sth_and_uid` are more appropriate for the latter purposes, since they more efficiently check for the file's prior existence when opening it.

4.7 How to Write a Deriver in C

To write a deriver in C, you need to know (in addition to the material in the previous sections), the precise types and calling conventions for the functions that constitute the deriver, how to manipulate the values with various relevant utilities, how to build an executable that can be installed as a worker, and how to debug your new deriver.

4.7.1 How to Create or Extend a C Worker Executable

As usual, we begin at the end, with a description of what to do once you have actually written your deriver. You then create a C worker, as a `Unix-program` artifact. For example:

Example C worker

———— *Targets* ————

sun4-dbg [cc (universal artifact)] **-g**
 C [cc (universal artifact)] **-g -I/e-l/lib/include**

———— *Modules/Instructions* ————

[Example C worker initialization module]
[Example C worker implementation]
-L/e-l/lib/sun4 -lworker -lantique -lmodern

A line-by-line explanation:

- The first line is the caption.
- The entry in the Targets section (which could, in general, be one of several) contains compilation switches that cause symbols to be saved for debugging (`-g`) and that guide the compiler to some header files outside the artifacts system (`-I/e-l/lib/include`).[5]
- The Modules/Instructions section refers to one module (in this case, the first) that contains the worker initialization function, whose name must be `extenders_init`. This parameterless, resultless function must include the initialization code for each tool in the worker. Normally, this is done by references to `C-source` artifacts, so that the actual code can also be referred to in documentation. This initialization module is written by the extender, but may be patterned after extant examples.

[5] In the future, all dependences on files outside the artifacts system will be represented via artifact references. E-L now has all the machanisms to make this possible, but not all of its source code has been converted to use them.

- The remaining modules (here only one) have the code for the various derivers in the worker.
- The final line consists of switches that cause the worker to be linked with objects that outside the artifacts system.[5]

The initialization module for the example worker registers its only deriver, which needs to supply only one function.

Example C worker initialization module

——— *Targets* ———

——— *Source* ———

[stdio.h]<spec>
[deriver.h]<spec>
[drvr-utils.h]<spec>
[Example C worker implementation]<spec>

global ⊥
```
  void
extenders_init()
{
  enter_deriver("my-type/example",
                NULL,           /* computable-kinds */
                NULL,           /* requisite-kinds */
                NULL,           /* reference-handle-kinds */
                NULL,           /* compare-relative */
                example_deriver,
                NULL,           /* abort-job-hook */
                1);             /* job-server-incremental (== TRUE) */
}
```

The example tool does nothing but produce a constant derivative.

Example C worker implementation

——— *Targets* ———

——— *Source* ———

global ⊥ [deriver.h]<spec>
[drvr-utils.h]<spec>

[*derive* function for `example` kind]

In a more complex deriver, there would be a reference to each of the deriver functions (e.g., the *compare-relative* function), and there would probably be supporting functions as well. The prerequisite artifacts [deriver.h] and [drvr-utils.h] supply necessary type and function declarations.[6] (See the next section for more on this subject.)

[6]At present, these are `universal` artifacts connecting to header files outside the artifacts system. When the conversion of E-L's source code is complete (see footnote 5), these will be replaced by references to ordinary module artifacts.

Finally, the one necessary function:

derive **function for** **example kind**

```
global  string_sequence_type *
example_deriver(dossier, relative_k_D_pairs, hint_list)
  dossier_type *dossier;
  k_D_sequence_type *relative_k_D_pairs;
  generic_pointer_sequence_type *hint_list;
{
  return singleton_derivative_sequence("some derivative");
}
```

See Section 4.6 for `singleton_derivative_sequence`.

4.7.2 Types and Calling Conventions

We first describe the correspondence between C types and those used in the language-independent description of derivers. For C types with "`sequence_type`" in their names and for "`string_sequence2_type`", see the chapter on sequences in *Artifacts System C Utilities*. The list below gives the type of an element of the sequence.

- *body*—`char *`
- *boolean ⊎ hint*—`boolean_or_generic_pointer *`; cast your hint, or the values `TRUE` or `FALSE`, to this type.
- *dossier*—`dossier_type *`, where the struct has the following types/fields:
  - `char *body`
  - `generic_pointer contents`
  - `k_d_sequence_type *requisite_k_d`
  - `h_k_d_sequence_type *reference_h_k_d`
- *handle-kind-derivative-sequence*—`h_k_d_sequence_type *`; elements have type `h_k_d_type`, which has fields:
  - `int handle`
  - `char *kind`
  - `char *derivative`
- *handle-kind-sequence*—`h_k_sequence_type *`; elements have type `h_k_type`, which has fields:
  - `int handle`
  - `char *kind`

- *hint-set*—`generic_pointer_sequence_type *`; elements have type `generic_pointer`.
- *integer*—`int`
- *kind-derivative-sequence*—`k_d_sequence_type *`; elements have type `k_d_type`, which has fields:
  - `char *kind`
  - `char *derivative`
- *kind-derivative-set-sequence*—`k_D_sequence_type *`; elements have type `k_D_type`, which has fields:
  - `char *kind`
  - `string_sequence_type *derivative_set`
- *kind-sequence*—`string_sequence_type *`; elements have type `char *`.
- *kind/uid-request-sequence* —

  ```
  typedef struct k_ur_sequence_type {
    k_ur_type *elements;
    int length;
  } k_ur_sequence_type;
  ```

  where elements have the type

  ```
  typedef struct k_ur_type {
    char *kind;
    int uid_count
  } k_ur_type;
  ```

- *location*—`generic_pointer *` (See *Artifacts System C Utilities.*)
- *uids*—`string_sequence2_type *`

The calling conventions for the C functions are exactly analogous to those in the language-independent specification, using the corresponding C types. The only slight subtlety involves *location* arguments, whose type is `generic_pointer *`. With both *location* and *hint*, the extender will likely cast the generic pointer to some more specific data type.

4.7.3 Relevant Utilities

The document entitled *Artifacts System C Utilities* contains many useful types and functions that are available to the author of tools in C. Chapter 2 of that document, "Tool Writer Facilities", has interface descriptions for the various utility functions (and other program objects) mentioned but not defined throughout this section. Read Chapter 1 for a instructions on incorporating the utilities into your derivers. (There should be no need to consult Chapter 3, "System Extender Facilities".) If you don't find what you need defined either in this document or Chapter 2 of *Artifacts System C Utilities*, either appeal to the editor of the latter or write it yourself.

Of particular interest are functions that allocate "job-persistent" storage, which is heap storage that is registered to be garbage-collected when the C worker returns to its lowest-level listening loop, i.e., when all jobs it has been engaged in are done. Since tools rarely need heap data that lives longer than the current job, the tool writer is freed from worries about reclaiming storage.

Chapter 2 also contains important information about exception handling, accessing derivative files, reading and manipulating artifact contents and using regular expression searches. There are also definitions of a variety of useful data structures, including sequences, hash tables, and standard text handles (variable-length arrays of characters).

The rest of this section describes utilities of only to C deriver writers. Some are mentioned in examples and algorithms elsewhere in this document. These functions must be used in the context of a `<spec>` reference to the `universal` artifact called `drvr-utils.h`. (See the example deriver above.)

- **`h_k_sequence_type *uniform_handle_kinds(first, last, kind)`**
  `int first, last;`
  `char *kind;`

- **`char *derivative_path(kind, filename)`**
  `char *kind, *uid;`

- **`char *derivative_area(kind)`**
  `char *kind;`

  - See the corresponding language-independent functions in Section 4.6.

- **`FILE *stream_and_uid(flag, area, extn, uid)`**
  `int flag; char *area, *extn, **uid`

  - See `stream-and-uid`.
  - The result is *stream*.
  - `uid` must be the non-null address of a `char *`, in which `sth_and_uid` places the *uid* result of `stream-and-uid`.

- **`char *filename_and_uid(flag, area, extn, uid)`**
  `int flag; char *area, *extn, **uid`

  - **filename_and_uid** is to **filename-and-uid** (section 4.6) as **stream_and_uid** is to **stream-and-uid**.
  - Use this function should only if you want the filename for purposes other than simply opening a stream on it. The functions **stream_and_uid** and **sth_and_uid** (below) are more appropriate for the latter purposes, since they more efficiently check for the file's prior existence when opening it.

- **sth_type sth_and_uid(flag, area, extn, uid)**
  int flag; char *area, *extn, **uid
  - Returns, in gc'ed storage, an **sth_type** in a directory determined by **flag** and **area**. The other arguments are as for **stream-and-uid**.

- **char *new_derivative_filename(kind, extn)**
  char *kind, *extn;
  - OBSOLETE. Use **filename_and_uid**.
  - For backward compatibility, if **extn** is DEFAULT_EXTN then the filename will have any file extensions appropriate for the **kind**. This usage is deprecated.
  - Returns the uid corresponding to **filename** in newly allocated, gc'd storage The uid is a string representing a decimal integer.
  - Returns NULL if it can't find a uid in **filename**.
  - Simply strips off UNIX filename extensions and checks that what's left looks like a uid. Although generality might suggest otherwise, for sanity checking it assumes that the extension begins with and includes the first period ('.') in **filename**.

- **int chwd(oldpath, newpath)**
  char *oldpath, *newpath;
  - If **newpath** is not NULL, changes the current UNIX working directory to **newpath** using **chdir**.
  - Puts the previous working directory (obtained with **getwd**) into **oldpath**, unless **newpath** is NULL, when it does nothing.
  - The caller is responsible for allocating enough storage (at least MAXPATHLEN characters) for **oldpath**.
  - Returns 0 on successful completion, or -1 if **newpath** is null.
  - Signals an error if **chdir** encounters a problem.

- **string_sequence_type *singleton_derivative_sequence(derivative)**
  char *derivative;
  - Returns a sequence with exactly one string element, a newly allocated, automatically garbage-collected copy of **derivative**.

-
  ```
  boolean same_file_contents(path1, path2, directory)
  char *path1, *path2, *directory;
  ```
  - Returns `TRUE` if the contents of the files given by `path1` and `path2` are identical.
  - The path names are taken relative to `directory` if it is not `NULL`. If `directory` is `NULL`, the path names are taken relative to the current working directory (unless they are fully qualified).
  - Using `directory` or path names relative to the current working directory (rather than fully qualified path names) considerably speeds up the comparison.
  - For example, to compare two derivatives in `uid1` and `uid2` of kind `manuscript`, use

    ```
    same_file_contents(uid1, uid2, derivative_area("manuscript"))
    ```
  - Signals an error if either file does not exist.

-
  ```
  int_sequence_type *related_references(old_dossier, old_handles,
                                         new_dossier, new_handles)
  dossier_type *old_dossier, *new_dossier;
  int_sequence_type *old_handles, *new_handles;
  ```
  - Compares subsets of the references of two related artifacts, returning a map from the reference subset of the younger artifact. The map indicates which member of the older artifact's reference set each younger-artifact reference is related to, if any.
  - The inputs are dossiers for the two artifacts and sequences containing handles relative to each. The output is a sequence of integers parallel to the `new_handles` sequence; each entry is either a handle from the sequence `old_handles` or `-1`, which means there's no relationship.
  - An artifact is related to another if 0 or more artifact predecessor links lead from the first to the second. Thus, among other instances an artifact is related to itself, and also to its immediate predecessor (if there is one). Notice that the definition is not symmetrical — for the purpose of this function, relatedness goes backward in time.

-
  ```
  char *compare_files(new_file, file_set)
  char *new_file;
  string_sequence_type *file_set;
  ```
  - Compares the file named by `new_file` to each of the files in `file_set`.
  - The files named may be a mix of paths relative to the current working directory or absolute, fully-qualified path names. However, since the mention of any absolute path names seriously affects efficiency, it is best to confine all the files to paths relative to the current working directory.

  - If some element of **file_set** has byte-by-byte the same contents as **new_file**, deletes *new_file* and yields that element. Otherwise, yields **new_file**.
  - The string returned is shared either with **new_file** or with one of the elements of **file_set**.
  - In typical use the **new_file** was just created in the process of derivation. The **file_set** names old files which are derivatives of relatives.
  - Uses **same_file_contents** to do the comparisons.

- **string_sequence_type * derivative_set_new_gc**

  - Return a new, empty, initialized set of derivatives in gc'd storage.

- **string_sequence_type ***
  select_derivatives(k_D_seq, $kind_1$**,** d_1**, ...,** $kind_n$**,** d_n**, (char*)NULL)**
  kind_derivative_set_sequence_type *k_D_seq;
  char *$kind_1$**, ***d_1**, ..., ***$kind_n$**, ***d_n**;**

  - Return a subsequence of the d_i that is parallel to the kinds in **kind_derivative_set_sequence**.
  - **kind_derivative_set_sequence** has type **kind_derivative_set_sequence ***
  - Each k_i and d_i is a char*; *each* k_i *is a derivative kind and each* d_i *is the corre*sponding derivative.

- **k_ur_sequence_typek_ur_seqh_make_gc(n,** $kind_1$**,** $count_1$**, ...,** $kind_n$**,** $count_n$**)**
  int n, $count_1$**, ...,** $count_n$**;**
  char *$kind_1$**, ..., ***$kind_n$**;**

  - Construct a kind/uid-request sequence of length n using each **kind**$_i$ as a kind and each **count**$_i$ as the corresponding uid-request count.

- **area_type** One of TYPE or KIND. Values are used by the following function to distinguish between artifact contents files (TYPE) and derivative files (KIND).

- **char* resource_path(nature, subarea, filename)**
  area_type nature; char* subarea, *filename;

  - Construct the repository-relative path to **subarea/filename**, given the **nature** of **subarea**, which is either a TYPE string or a KIND string. A repository-relative path is one whose implicit root directory is that given in the environment variable E_L_REPOSITORY.

4.7.4 How To Install A Worker Developed in E-L

Here is a recipe for creating a new worker from a `unix-program` artifact developed as described in section 4.7.1.

1. Determine the kind of derivative to export and install by selecting one of the target lines of the program artifact. If its *machine-variant* combination is `sun4-dbg`, as in the example of section 4.7.1, then the derivative kind is `executable%sun4-dbg`.
2. Choose a directory in the file system to hold the executable, and a file name for it. Together these form a fully qualified path name at which the worker will be stored.
3. If the tools in your worker develop new classes of derivatives, make appropriate entries in the `derivative-classes` plex, as described in section 3.2.
4. Enter the tools of the worker in the `derivers` plex (section 4.4).
5. Select a logical name for your worker. It need not be the same as the file name under which it is stored. Make an entry in the `workers` plex, using the logical name and the fully qualified path name from step 2. See section 4.3 for more details concerning the `workers` plex.
6. From E-L, use the `M-x export-derivative` command to export an executable of the kind selected in step 1. The command will prompt you first for the program artifact, and then for the file name where you want it to reside. Give it the fully qualified path name from step 2.[7]
7. Use the UNIX command `chmod` to make the newly created file executable.

Your worker should now be installed and ready to handle jobs.

4.7.5 Debugging A Deriver

This section describes some strategies, tools, and pitfalls associated with debugging a deriver offered by a C worker. Ultimately, the methods described here will be superseded by an approach that is more integrated with E-L.

Here in outline is the basic procedure for debugging a worker. Details follow.

1. Follow the steps in section 4.7.4 to create the worker executable and export it to the UNIX file system.
2. Set up the `derivers` and `workers` plexes to specify the worker you wish to debug.
3. Using `M-x dbx` within E-L, or using your favorite debugger outside the system, start a debugging session on your worker.

[7]Notice that, despite its name, the `export-derivative` command is not fully general. It applies only to certain derivatives that have files associated with them, e.g. the `executable` derivatives of `unix-program` artifacts. For other artifact types and derivative kinds, beware.

4. Start a job server as an inferior process to the worker. This job server will schedule jobs for the test worker.
5. Commit artifacts or use `M-x derive-from` to schedule derivations on the test worker from your E-L session. You may watch progress in the `present` plex.
6. When you are done debugging, shut down the job server from the worker, and exit the worker and debugger.

If your worker handles derivatives also handled by existing workers, and there are others using your repository, your jobs may interfere with those scheduled by others. For example, a derivation you scheduled to test your worker may go to another worker, or your worker may get jobs scheduled by others. Shutting down competing job servers may help, but it may also make other users unhappy. It's best to test only when the system is otherwise quiet.[8]

Notation. We use `%` as the UNIX shell prompt and `(dbx)` for the debugger prompt.

Invoking the debugger. A worker developed in C within E-L can be debugged within E-L;[9] just apply the `M-x dbx` command to the `unix-program` artifact for the worker.

On the other hand, you can also debug a worker outside of E-L. To provide the right environment for executing a worker, however, you must use the `e-l` script to start the debugger. If your worker is `my-worker`, execute the following shell command:

```
% e-l -do dbx my-worker
```

(The debugger we use in the examples is `dbx`, but similar remarks apply to, say, `gdb`.)

Next, you should set breakpoints at strategic places in your code, for example at each top-level derivation function. You may also set breakpoints at the following functions to catch faults or exceptions raised by the `error` and `warning` functions:

```
(dbx) stop in server_fault
(dbx) stop in server_error
(dbx) stop in server_warning
```

Once you have set your debugging environment, execute

```
(dbx) run -i
```

to start the worker with its standard input, output and error output set to your terminal. (The `-i` switch tells the worker nucleus to parse RPCs on a one-per-line basis, so that you can drive the worker interactively. In normal operation, RPCs are packaged for efficient transmission between processes.)

[8]There will soon be an easy way to set up additional repositories to allow debugging without interference.

[9]So it is unfortunate that you need to export the program before debugging begins simply in order to register it in the `workers` plex. This will be changed soon. For now, you can export once before testing your worker, then debug within E-L, and finally export one more time to put your final worker into production.

Starting an inferior job server. Once your worker is started, you need to start a job server that will send derivation jobs to your worker. This is done by starting a job server inferior to the worker. If you have started the worker as in the previous paragraph, it will be waiting for input from your terminal. The worker interprets each line you enter at your terminal as an RPC (remote procedure call). The first RPC you send starts the inferior job server. Enter the line

```
    start-server (worker-logical-name host)
```

followed by a carriage return. The *worker-logical-name* is the first entry in the current worker's `workers`-plex line (its logical name, not its executable file name), enclosed in double quotes. *¡host¿* is the host name (also enclosed in quotes) of the machine on which you are executing the worker. So if the host is named `sam` and your worker has logical name `e-l-latex-worker`, you would enter the line

```
    start-server ("e-l-latex-worker" "sam")
```

After you've entered the RPC, the worker will type out the process id of the job server and wait for further RPCs. Remember this "pid" for later. You should soon see a new entry for job server in the `present` plex of your E-L session.

To stop a server without terminating the worker, enter the following line to the waiting worker:

```
    stop-server
```

The `stop-server` RPC has no arguments—it already knows about the inferior job server. If you are waiting at a debugger prompt, either because of an error or breakpoint, you can shut down the worker by having the debugger execute the following call:

```
    stop_server_receiver(0,0)
```

When all else fails, call the parameterless C function `kill_server()` from the debugger prompt, or try a `kill -2` on the job server's pid from the shell; these shut down an inferior server, and cause any jobs it is working on to be rescheduled.

To stop a worker in its idle loop (i.e., while it is waiting for an RPC from your terminal) issue the

```
    stop-worker
```

RPC. This gracefully shuts down any inferior job server and then exits the worker program back to the debugger.

4.8 How to Write a Deriver in Common Lisp

This section assumes familiarity with the `clisp-module` artifacts as detailed in the *Language Users' Manual.* To add a deriver to a new or existing Lisp worker, perform the following steps:

1. Write the six deriver functions described in section 4.5.
   - Section 4.8.2 describes the Common Lisp versions of data types mentioned in 4.5 and calling conventions for the deriver functions.
   - Section 4.8.3 describes various utilities useful for writing derivers.
2. Extend an existing Lisp worker with your new deriver, or create a new one. See section 4.8.1 for details.
3. Debug your deriver using methods described in section 4.8.4.

4.8.1 How to Create or Extend a Lisp Worker Executable

A Lisp worker is a UNIX executable obtained from a `clisp-program` artifact of the appropriate form. Here is an example (in fact the only Common Lisp worker currently supported—they are quite large.)

——————— *Targets* ———————

`sun3lucid4` [Lucid 4.0.1] **`e-l/::standalone-lisp-worker`**

——————— *Modules/Instructions* ———————

[Root module of Common LISP worker]

The single `clisp-module` referenced by the program above is called `lw-root`. The extender can either edit this module, and thereby alter the existing Lisp worker, or copy it to create another one by embedding it in a different `clisp-program`. This root module deals with some bootstrapping issues that need not concern the extender. The two points to note here are: where to insert a reference to the module containing a new deriver, and what to do about the Lisp package used to encapsulate that deriver. Here's the root module:

Root module of Common LISP worker

——————— *Targets* ———————

`sun3lucid4` [Lucid 4.0.1] [Safe Compiling Switches]

——————— *Source* ———————

```
(in-package "E-L")
```

[Utilities for bootstrapping foreign (C-coded) object files]

```
(let ((*load-if-source-newer* :compile)
      (*load-if-source-only* :compile))
```

  [Kernel for any Common LISP worker]<spec>
  [Debugging support for Common LISP derivers]<spec>
  [The **clisp-module/pfasl** deriver]<spec>

  (Other deriver modules could be referenced here.)

  [Rename worker-kernel and deriver packages])

[Create a worker executable when bootstrapping]

It's probably a good idea to give each Common Lisp deriver its own package. Certainly the deriver **should not** be in package `USER`. Your package name should begin with `E-L`, since such package names are reserved to the worker kernel and derivers. Packages `E-L`, `E-L/`, `E-L-CLISP`, and `E-L-CLISP/` are already used: `E-L` is the package for the code of the Lisp worker kernel, and `E-L-CLISP` is that of the `clisp-module/pfasl` deriver. In each case, the source package is renamed by appending a / when the worker is built, as follows:

Rename worker-kernel and deriver packages

```
(when (find-package "E-L/") (delete-package "E-L/"))
(rename-package "E-L" "E-L/")
(when (find-package "E-L-CLISP/") (delete-package "E-L-CLISP/"))
(rename-package "E-L-CLISP" "E-L-CLISP/")
```

This package renaming is important to avoid confusion when deriving from the artifacts that are the source for the worker kernel and the deriver itself. If your deriver operates on Common Lisp code, and might therefore operate on its own source, you should probably rename its package in a similar way.

The function that registers the deriver in the worker can occur in the module that contains it; for example, [The `clisp-module/pfasl` deriver] referenced in the root module above contains the following definition:

Register deriver functions for the `pfasl` deriver.

```
(enter-deriver
  (format nil "clisp-module/pfasl-~A~A"            Concoct the deriver name
          (get-machine)
          (string-downcase (lisp-architecture)))
  'pfasl-f1  computable-kinds function
  'pfasl-f2  requisite-kinds function
  'pfasl-f3  reference-handle-kinds function
  'pfasl-f4  compare-relative function
  'pfasl-f5  derive function
  nil        abort-job-hook
  t)         job-server-incremental flag
```

Observe that the deriver functions are supplied as symbols. When the deriver is executed, the function values of these symbols are used. This is much better for debugging than entering actual function values, because tracing requests and or redefinitions act on symbols, not on function values.

4.8.2 Types and Calling Conventions

This section describes the correspondence between Common Lisp types and those used in the language-independent description of derivers in section 4.5. Recall the Common Lisp has typed *objects*, not typed variables.[10] Thus the specifications below mean that the objects passed to and returned from the relevant functions should have the indicated types.

The calling conventions for the Lisp functions are exactly analogous to those in the language-independent specification found in section 4.5, by way of the type correspondence below.

Recall that the Lisp type encompassing all objects is called `t`. In the following list we use standard Common Lisp type specifiers[11] whenever possible.

- *body* — Lisp type `string`.
- *boolean* ⊎ *hint* — We here define a boolean to be one of the values `nil` or `t`. A hint can be anything the deriver writer likes except one of these values. Thus the union type is, technically, `t`.
- *dossier* — A dossier is a `list` with the following accessors, each of which takes a single *dossier* argument and returns the type indicated:
  - `dossier-body` — Lisp type `string`.
  - `dossier-contents` — Lisp type `t`.
  - `dossier-requisite-k-d` — a *kind-derivative-sequence* (see below).
  - `dossier-reference-h-k-d` — a *handle-kind-derivative-sequence* (see below).

[10]See Chapter 2 of *Common Lisp The Language*, 2nd edition, for details.
[11]ibid., Chapter 4.

- *filename* — Lisp type `string`.
- *kind* — Lisp type `string`.
- *location* — a `list` with a single element, that element being the contents of the artifact under derivation. Unless changed by the deriver, the element is `nil`. Given a location `loc`, the deriver writer should access its contents using `(car loc)` and set the contents using `(setf (car loc) ...)`.
- *handle-kind-derivative-sequence* — Lisp type `(list h-k-d)`. Each `h-k-d` is a list of three elements

  (*handle kind derivative*)

  where *handle* has type `fixnum` and both *kind* and *derivative* have type `string`.
- *handle-kind-sequence* — Lisp type `(list h-k)`. Each `h-k` is a list of two elements

  (*handle kind*)

  where *handle* is a `fixnum` and *kind* is a `string`.
- *hint-set* — `list`. This requirement that a set be implemented as a list may be odious to some, and is subject to revision.
- *integer* — `fixnum`
- *kind-derivative-sequence* — Lisp type `(list k-d)`, where each `k-d-set` is list of two `strings`

  (*kind derivative*)
- *kind-derivative-set-sequence* — Lisp type `(list k-d-set)`. A `k-d-set` is a list of two elements

  (*kind derivative-set*)

  where *kind* is a `string` and *derivative-set* has type `(list string)`. Thus to obtain the *derivative-set* of a `k-d-set`, use `cdr`.
- *kind-sequence* — `(list string)`.
- *kind/uid-request-sequence* — `(list k-ur)` A `k-ur` is a list of two elements

  (*kind count*)

  where *kind* is a `string` and *count* is a `fixnum`.
- *path* — Lisp type `string`.
- *uids* — Lisp type `(list (list string))`

4.8.3 Relevant Utilities

Section 4.6 describes facilities for use by the deriver writer and provided by all workers. This section describes additional facilities, or clarifications to, or deviations from the specification in section 4.6. In general, if section 4.6 describes a function as returning more than one value, then the Lisp version returns multiple values whenever the Lisp dialect provides for them; otherwise the function returns a list of values.[12]

`enter-deriver` *[function]*

Args: *tool computable-kinds requisite-kinds reference-handle-kinds compare-relative derive abort-job &optional (job-server-incremental t)*
Result type: no result

- Register for *tool* the six functions defining a deriver in the current Lisp worker.
- *tool* should be a `string`. The remaining arguments should be symbols, `nil` if the default functions are to be used.
- Obviously, *derive* cannot be `nil`.

`stream-and-uid` *[function]*

Args: *flag area &optional (extension nil)*
Result type: `stream string`

- See section 4.6 for the semantics.
- *flag* is one of the keywords `:derivative` or `:artifact`, rather than an integer.
- The result is a Lisp output `stream` and a `string` giving the uid.

`filename-and-uid` *[function]*

Args: *flag area &optional (extension nil)*
Result type: `stream string`

- See section 4.6 for the semantics.
- *flag* is one of the keywords `:derivative` or `:artifact`, rather than an integer.
- The result is two Lisp `strings` giving the file name and the associated uid.

`bad-derivative` *[function]*

Args: *message*
Result type: `string`

[12]Common Lisp implementations return multiple values.

- Return a properly constructed bad derivative encoded with the given `string` *message*.

`error-derivative` [*function*]

Args: *summary type details uid-list*
Result type: `string`

- Return an error derivative using the `strings` *summary*, *type*, *details* and the list of uids in *uid-list*.
- See section 3.5 for a description of the syntax and treatment of error derivatives.

`artifact-contents-path` [*function*]

Args: *type &optional filename*
Result type: `string`

- See the corresponding definition in *Artifacts System C Utilities* for the semantics.
- *type* and *filename* are Lisp `strings`.

`with-stream-and-uid` [*macro*]

Args: *(stream uid flag area &optional (extension nil) (abort nil)) &body body*
Result type: None

- Bind *stream* and *uid* as the results of a call to `stream-and-uid`, and execute *body* in a manner analogous to the Common Lisp macro `with-open-file`.
- If the boolean *abort* is not null, then delete the file bound to *stream* upon abnormal exit. Otherwise just close it.
- The arguments *flag*, *area* and *extension* have the same meaning that they do in `stream-and-uid`.
- Because of the unwind-protection one should prefer this macro over direct uses of `stream-and-uid`.

4.8.4 Debugging A Deriver

Debugging a Common Lisp worker turns out to be rather different from debugging the other workers. The main issue is that the Common Lisp worker is multi-threaded.

Initially you will be debugging a Common Lisp worker assembled in an interactive Lisp Session. Suppose the worker name, as registered in the `workers` plex, is `sun4lucid4`, and that that host you're running it on is `myhost`. You can start an inferior job server under your interactive worker by calling `usual-start-server` with the strings `"sun4lucid4"` and `"myhost"` as arguments. To interrupt the worker's main event loop, use a normal `^C` keyboard interrupt. At that point you can stop or kill the inferior server by invoking `stop-server` or `kill-server`, respectively.

However, any time the worker enters a break, you must know which task is currently executing. If the worker is at its initial task, it is safe to choose the option of returning to top level. However, if a task associated with a derivation is currently executing and you don't want to continue, you should usually select the *kill current process* option. (By "process", Lucid means a Lucid task.) Then you will end up with the dispatching process still running.

You can debug a standalone worker interactively. To do so, start the executable with a single Unix command-line argument: `-interactive`. When the worker starts up you will get an interactive Lisp prompt, and can use `usual-start-server`, and so on.

4.9 How to Write a Deriver in EMACS LISP

To add a deriver to a new or existing EMACS worker, perform the following steps:

1. Write the deriver functions described in section 4.5.
   - Section 4.9.2 describes the EMACS Lisp versions of data types mentioned in 4.5 and calling conventions for the deriver functions.
   - Section 4.9.3 and the document *Emacs Lisp Utilities* describe various utilities useful for writing derivers.
2. Extend an existing EMACS worker with your deriver, or create a new worker. See section 4.9.1 for details.
3. Debug your deriver using methods described in section 4.9.4.

4.9.1 How to Create or Extend an EMACS Worker Executable

To create a new EMACS worker it is necessary to provide an EMACS Lisp file which

- includes the top-level form `(require-emacs-lisp "elw")`,
- defines the nullary function `extenders-init`,
- and in this function calls `enter-deriver` with the names of the functions that constitute your deriver.

Section 4.9.1 presents a simple example.

To extend an existing worker, it is only necessary to augment the `extenders-init` function for that worker with a call to `enter-deriver` to make your new deriver known to the worker.

- **enter-deriver** *deriver-name* f_1 f_2 f_3 f_4 f_5 f_6 *job-server-incremental* →
  - Register the deriver named *deriver-name*. See page 4-8 for details.
  - At present f_6 must be `nil`.

It is also necessary to add a new target entry to the following `Unix-program` artifact

Emacs Lisp Worker Monitor

```
———————— Targets ————————————————————————————————
sun4-dbg-manuscript [cc (universal artifact)] -g -Bstatic
 C [cc (universal artifact)] -g -DLOAD_FILE_NAME="MS-derivers"
mips-dbg-manuscript [cc (universal artifact)] -g
 C [cc (universal artifact)] -g -DLOAD_FILE_NAME="MS-derivers"
———————— Modules/Instructions ———————————————————
[elw-parent]
```

This program starts the EMACS process by loading `basic` and a single file provided by you, and causes EMACS to execute the function `elw:main`. In addition, Emacs Lisp Worker Monitor manages the standard output and error of the EMACS process, causing RPC's to be sent out on standard output, causing other messages to be sent out on standard error (whence they end up in the job log). If all the functions for your derivers are loaded by loading the file `example.elc`, then you would add targets (one for each architecture) to Emacs Lisp Worker Monitor, each of which compiles it with the switch

```
-DLOAD_FILE_NAME="example"
```

The result would be:

Emacs Lisp Worker Monitor (with new, `example` deriver added)

```
———————— Targets ————————————————————————————————
sun4-dbg-manuscript [cc (universal artifact)] -g -Bstatic
 C [cc (universal artifact)] -g -DLOAD_FILE_NAME="MS-derivers"
sun4-dbg-example [cc (universal artifact)] -g -Bstatic
 C [cc (universal artifact)] -g -DLOAD_FILE_NAME="example"
mips-dbg-manuscript [cc (universal artifact)] -g
 C [cc (universal artifact)] -g -DLOAD_FILE_NAME="MS-derivers"
mips-dbg-example [cc (universal artifact)] -g
 C [cc (universal artifact)] -g -DLOAD_FILE_NAME="example"
———————— Modules/Instructions ———————————————————
[elw-parent]
```

Example EMACS Deriver The EMACS code for the example deriver could be the following:

```
(require-emacs-lisp "elw")                              emacs-example
```

```
(defun extenders-init ()                                          emacs-example
  (enter-deriver "example-k" nil nil nil nil 'example-derive nil t))

(defun example-derive (dossier relative-k-D-pairs hint-list)     emacs-example
  (vector "dome derivative"))
```

4.9.2 Types

We enumerate the EMACS Lisp representations of the language-independent types used above.

- *body*—`string`
- *boolean* ⊎ *hint*—Arbitrary lisp value; the symbol `t` is boolean true, the symbol `nil` is boolean false, and any other value is a hint.
- *cell*—list, with slot access functions. The functions `cell-`*slot* take a cell as argument and return the value of *slot*. Valid values for *slot* are
  - `pointer`
  - `body`
  - `references`
  - `timestamp`
  - `type`
  - `name`
  - `creator`
- *derivative-string*—`string`
- *dossier*—list, with slot access functions. The functions `dossier-`*slot* take a dossier as argument and return the value of *slot*. Valid values for *slot* are
  - `body`
  - `contents`
  - `requisite-k-d`
  - `reference-h-k-d`
  - `cell`

  The cell used to be private, but too many derivers required access to it.
- *handle-kind-derivative-sequence*—a vector of lists, each of the form

  (*handle kind derivative-string*).

- *handle-kind-sequence*—a vector of lists, each of the form

  (*handle kind*).

- *hint-set*—a list of arbitrary values, excluding the symbols `t` and `nil`.
- *integer*—**`integer`**.
- *handle*—**`integer`**.
- *kind*—**`string`**.
- *kind-derivative-set-sequence*—an a-list from *kind* (string) to a list of derivative strings.
- *kind/uid-request-sequence*—a vector of lists, each of the form

  (*kind integer*).

- *location*—a cons cell; **the deriver must modify only the car; never the cdr.**
- *uids*—a vector of vectors of strings.

4.9.3 Relevant Utilities

See the *Emacs Lisp Utilities* document for general-purpose utilities. Otherwise there is only a single deriver-specific utility:

- **`filename-and-uid`** *flag area &optional (extn nil)* → (*filename . uid*)
  - See section 4.6.

4.9.4 Debugging A Deriver

An EMACS worker is debugged in an interactive EMACS session, from which an inferior job server can be started.

The connection to the job server (while debugging) is fairly fragile. The most difficult problem is that EMACS Lisp can't tell the difference between the standard error and the standard output of the job server. Therefore when the job server writes log messages, they go to the worker's RPC input. This is not so bad, because the worker understands these log messages as RPCs, and acts appropriately. The problem arises when the job server writes a log message while processing an RPC that has a reply. In this case the worker interprets the log message as an (illegal) reply to the RPC. There seems to be no way around this without augmenting the standard RPC mechanism.

Messages from both the worker and the job server appear in the buffer named `*elw-log*`.

To debug an EMACS worker we recommend the following procedure:

1. Make sure that the `derivers` and `workers` plexes are set up correctly.

2. Using `e-l -do`, start an EMACS (not Epoch) session.
3. Load the EMACS Lisp file `basic` (with `M-x load-file $E_L_BIN/emacs-lisp/basic`), and then load the EMACS Lisp file that defines the derivers (which must `require-emacs-lisp elw`).
4. Issue the command `M-x init` (which sets `debug-on-error` to `t`).
5. Issue the command `M-x start-server` to start an inferior job server. When prompted, enter the name of the worker being debugged (from the `workers` plex), and a name for the current host that is associated with the desired architecture in the `hosts` and `workers` plexes.

   This also starts the worker's main event loop; you must use `C-g` to quit this loop.
6. The command `M-x stop-server` stops an inferior job server that has no jobs.
7. The command `M-x cleanup-servers` kills (with `SIGINT`) any job servers that the worker still knows about.
8. There is also a `M-x kill-server` command, which will kill the most recently started job server (if it still exists).

A Thumbnail Sketches of Umpteen Derivers

The sections below have titles of the form t, k possibly followed by a qualifier. Such a section describes the functional arguments given to the `universal_deriver` to compute from an artifact of type t a derivative of kind k. The artifact under derivation will be denoted p, and in *compare-relative*, the relative is p'.

Each section below gives a brief sketch of a deriver, including, when k is a single derivative class, the representation of the derivative class (as seen in the `derivative-classes` plex), and in all cases the *kind-re* and *kind-re-replacement* (as seen in the `derivers` plex) and a description of the functional arguments to the `universal-deriver`. These summaries are not meant to stand alone, but to facilitate comparisons of different derivers, and to suggest strategies for a deriver you might want to write.

A.1 t, `caption` for Many Types t

See the document LaTeX *Artifacts* for a more complete discussion.

- *representation*: `[^\0\1\12]*`
- *kind-re*, *kind-re-replacement*: `caption`
- *computable-kinds*: null, because `caption` is the only kind.
- *requisite-kinds*: null, no derivatives necessary from p.
- *reference-handle-kinds*: null, because the caption does not depend upon references.
- *compare-relative*: null, because with a null set of handle/kind pairs, this would be called only when the bodies are different, in which case, nothing is known about the caption.
- *derive*: read the body, and for most types, use the first line, perhaps stripping off some comment characters; ignore derivatives of predecessors, because there are no uid's to be re-used.

A.2 t, `manuscript` for Many Types t

This section applies to artifacts whose contents are set in `\tt` font, except for its captions.

- *representation*: See section A.3 for `manuscript` representation. In the cases considered here, the derivatives will have the representation `L`f, where f. is a file in the `manuscript` directory containing the LaTeX source.
- *kind-re, kind-re-replacement*: `manuscript`
- *computable-kinds, requisite-kinds*: null
- *reference-handle-kinds*: `uniform-handle-kinds(0,` $n-1$`,caption)`.

- *compare-relative*: null, because all of the body and all of the references' derivatives affect the `manuscript` derivatives.
- *derive*: obtain a new uid f, open a stream on it, and:
  - Output some initialization to the stream that will establish a scope in which page breaks will not occur.
  - Output `\tt` and a newline to the stream.
  - Output the body to the stream, quoting all special LaTeX characters.
  - When encountering a reference, output its **`caption`**, surrounded by braces and a change of the font to **`\rm`**.
  - Output the close of scope to the stream, close the file.
  - Let the *kind-derivative-set-sequence* parameter be $\{\langle\texttt{manuscript}, \texttt{L}f'\rangle\}$, and let f'' be the result of `compare-files`(f, f'). The result is $\{\langle\texttt{L}f''\rangle\}$.

A.3 latex-piece, manuscript

See the document LaTeX *Artifacts* for a more complete discussion.

- *representation*: This is wildly complicated, to provide for efficient incremental derivation in highly tree-structured documents. It is included only for completeness, and to show how complicated derivatives can be:

```
'Lf..                         A LaTeX source file (ending with dot)
|'Pf..'  [^\0\1\12]*          A PostScript file and a macro string
|'Bb{latex-piece}..           A latex-piece body.
|'N(:lnode:s..<               A file having ...
 b{latex-piece}..             a latex-piece body, and
 ('␣                          a list of space followed by
  ('C[^\0\1\12]*'\1           a caption, or
  |'Lf..                      a LaTeX source file, or
  |'Bb{latex-piece}..         a latex-piece body, or
  |'P[^\0\1\12]*'\1           a macro string,
  |'N=lnode=)                 recursively, a node file.
 )+>)                         End list in the file.
 ('/:pnode:s<                 Optionally, another file with
  ([0-9]+                     a part index
   (':f..                     a PostScript file, or
   |'/=pnode=)                a PostScript node.
  )+<' >>                     End space-separated list, end file.
 |)                           The no file option.
```

- *kind-re*, *kind-re-replacement*: `manuscript`

- *computable-kinds*, *requisite-kinds*: null
- *reference-handle-kinds*: The kinds here are not quite uniform—for each handle, the kind is **`manuscript`**, unless the artifact specifies **`<caption>`**, in which case it is, surprise, **`caption`**.

There are useful hints for this deriver: a uid of an **`lnode`** (the LaTeX part) or the uid of a **`pnode`** (the PostScript part). (If a single predecessor has both, its derivative is good for p.)

- *compare-relative*:
  - Let the *kind-derivative-sequence* argument be $\{\langle$`manuscript`$, m'\rangle\}$. If m' is not of the form `N`n'`/`r', yield false (for reasons too subtle to go into here).
  - If p' and p have the same bodies and if the "LaTeX part" of the derivatives of references of p and the contents of n' are sufficiently similar, yield n' as the hint.
  - If "PostScript part" of the derivatives of references of p and the contents of r' are sufficiently similar, yield r' as the hint.
  - Otherwise, yield false.
- *derive*:
  - If the result can be constructed by using pieces of hints, that is the result.
  - Let b be the body of p.
  - If p has no references, the result is $\langle$`B`$b\rangle$.
  - If there is an n among the hints, it will be the n part of the result. Otherwise, get a new uid n (in **`manuscript`**) for an **`lnode`**. The first item in its contents will be b. Construct the rest of the contents of n from the derivatives of references of p, and while doing so, keep track of any PostScript that is encountered.
  - If no PostScript parts were encountered, the result is $\langle$`N`$n\rangle$.
  - Otherwise, if there is an r among the hints, it will be the r of the result. If there is no such r, get a new uid r (in **`manuscript`**) for a **`pnode`**, and fill in its contents using the PostScript encountered in the previous step. The result is $\langle$`N`n`/`$r\rangle$.

A.4 latex-root, latex-directory

- *representation*: Not quite as bad as **`manuscript`**:

```
d'␣[0-9]+                                A directory and a magic integer.
('␣                                      Beginning of possible list.
 :pcontents:                             Contents of a pnode.
 (([0-9]+                                a handle index
   (':f{manuscript}..                    a PostScript file, or
  |'/s{manuscript}<=pcontents=>)         a PostScript node.
 )+<' >)                                 End space-separated list.
|)                                       End of possible list and null option.
```

- *kind-re*, *kind-re-replacement*: `latex-directory`
- *computable-kinds*, *requisite-kinds*: null
- *reference-handle-kinds*: for most handles, the **`manuscript`** is needed; exceptions are the x derivative for references marked with `<`x`>`, for x = `bibliography` and `caption`.

A hint for this derivation consists of the uid of an existing derivative, a flag saying whether the bibliography information is valid for p, map from new part indices to old part indices, a vector of flags saying whether the new part is "LaTeX-equivalent" to the old, and a flag saying whether everything except the PostScript information is the same.

- *compare-relative*:
  - Look at the bodies of p' and p, and the derivatives of bibliographic references to compute the bibliography flag.
  - To obtain the map from the new to old parts, determine historical correspondences between the references of p' and those of p, where in each case the references are those after the `\begin{document}`. (The magic integer in the derivative makes locating these references easy.)
  - The vector of flags for LaTeX-equivalence is computed by looking at the body of p and the derivatives of references occurring after `\begin{document}`.
  - The all-ok-except-maybe-PostScript flag is true if and only if the map from new to old parts is the identity vector, the magic number for p' matches what would be the magic number for p, the bibliography flag is true, and the LaTeX-equivalence vector is all true.
- *derive*:
  - If there is a hint in which all that has changed is the PostScript, it is not necessary to set up a new directory. The new derivative string can use the old uid for the directory, and re-use any of the PostScript data, if possible.
  - Otherwise, set up a new directory *dir*.
  - If there is no hint, initialize *dir* and run LaTeX in it.
  - If there is more than one hint (oh lucky day), the best one is that in which the derivatives of the header references are identical. Other than this, a single hint is picked arbitrarily.
  - Initialize *dir* to just the differences indicated by the hint, and run LaTeX only where the hint indicates a necessity.

There is substantial omission of detail here. LaTeX is probably the most difficult tool to integrate.

A.5 `shellscript`, k for $k \neq$ `manuscript`

The **`manuscript`** derivative of a **`shellscript`** artifact is not obtained by running it, but as described in section A.2. For all other derivatives:

- *representation*: A **`shellscript`** artifact can produce any kind of derivative whose representation is `f`, `f<`*extension*`>`, or `d`. It also always produces `standard-output` and `standard-error` derivatives, whose representations are each `f`.
- *kind-re*, *kind-re-replacement*: `\^manuscript`
- *computable-kinds*:
  - Read the contents of the artifact
  - Yield {`standard-output`, `standard-error`} $\cup$ $\{k \mid$ `@`$\{k\}$ is in the contents$\}$
- *requisite-kinds*: null
- *reference-handle-kinds*: $\{\langle h, k\rangle \mid h\{k\}$ is in the contents$\}$
- *derive*:
  - Replace each $h\{k\}$ in the contents with the corresponding derivative, using the extension from the representation, if there is one.
  - Replace each `@`$\{k\}$ in the contents with string obtained by substituing a new uid for the `\f` or `\d` in the representation field of the derivative class of k.
  - Run the resulting shell-script directing standard output and standard error to new uids for `standard-output` and `standard-error`.
  - Let $\{\langle \textit{derive-}k_m, D_m\rangle\}_m$ denote the value of the *kind-derivative-set-sequence* argument and let $\{\langle k_j, d_j\rangle\}_j$ denote the *kind-derivative-sequence*. If any k_j is not among the *derive-*k_m, delete the file d_j. Otherwise, yield `compare-files`(d_j, D_m) as the result corresponding to *derive-*k_m.

Index

(list string), 4-28
elw-log, 4-34
-interactive, 4-31
:artifact, 4-29
:derivative, 4-29

abort-job-data, 4-11
accept-handle-from-artifact-buffer-*type*, *2-15*
accept-handle-from-draft-buffer-*type*, *2-18*
area_type (type), *4-21*
artifact pointer, *2-2*
artifact-body-uid-re-*type*, *2-9*
artifact-body-uid-re-*type*, 2-9
artifact-cell-pointer, 2-23
artifact-commands-*type*, *2-15*
artifact-edit-mode-*type*, *2-6*
artifact-edit-mode-*type*, 2-18
artifact-from-draft-lock-&-mode-list-*type*, *2-19*
artifact-type, 4-7
artifact-uid-from-draft-contents-*type*, *2-19*
artifacts (plex), 2-2
Artifacts System C Utilities, 4-30
auto-save form, *2-17*, 2-21
auto-save-draft-source, *2-22*
auto-save-draft-*type*, *2-21*
autoload, 2-4, 2-5
autoload-derivative-class, *3-3*
autoload-emacs-lisp, *2-5*
autoload-type, 2-4, *2-4*, 2-5

bad derivatives, *3-2*
basic (file), 4-32
bibliography (kind), A-4
body, *2-3*, 4-8, 4-9, 4-16, 4-27, 4-33
body-format, *2-11*, *2-19*
body-format-&-handles-that-appear-in-draft-*type*, *2-19*
body-format-&-handles-that-appear-in-draft-*type*, 2-20
body-format-&-handles-that-appear-in-header-module, *2-12*
body-format-&-handles-that-appear-in-header-*type*, *2-11*, *2-19*
boolean ⊎ hint, 4-27, 4-33
button-attribute-if-reference, 2-21

c-file (kind), 3-9, 4-9
c-module (type), 4-6
C-source (type), 4-14
c-source (type), 3-2, 3-9, 4-9
c-text (kind), 3-2, 4-9
caption (kind), A-1–A-4
cc, 3-8
cdr, 4-28
cell, 4-8, 4-33
chdir, 4-19
chwd, *4-19*
clisp-module (type), 4-25
clisp-program (type), 4-25
command, *1-2*
compare-files, A-2, A-5
compare-relative, 4-10, 4-11
compare_files, *4-20*
computable-kinds, 4-8
contents, *2-2*, 4-8, 4-9
copy-buttons-as-kill-hooks, 2-6, 2-18, 2-20
count, 4-28
create-draft, 2-17
create-draft-*type*, *2-17*
current-jobs, 4-1, 4-2

debug-on-error, 4-35
default-job-server-class-variant, 4-3, 4-4
defconst, 2-5
defvar, 2-5
delete-artifact-uid-*type*, *2-13*
delete-artifact-uid-*type*, 2-9
delete-derivative-class, *3-3*
delete-deriver, *4-7*
delete-host, *4-4*
delete-type, *2-2*
delete-worker, *4-5*
derivation, *1-3*
derivative, 4-28
derivative class, *3-1*
derivacive classes
    pfasl, 4-27
derivative string, *3-1*
derivative-area, *4-13*
derivative-classes (plex), 3-2–3-4, 4-12, 4-22, A-1
derivative-kinds-*type*, *2-6*
derivative-path, *4-13*
derivative-set, 4-28
derivative-string, 4-33
derivative_area, *4-18*
derivative_path, *4-18*
derivative_set_new_gc, *4-21*
derive, 4-10, 4-11
derive-from, 4-1
derive-*type/kind*, 4-2
derived-results, 4-10
deriver, *1-1, 1-2*
derivers, *4-7*
derivers (plex), 1-2, 1-3, 4-2, 4-3, 4-5–4-8, 4-22, 4-34, A-1
diff-artifact-*type*, *2-16*
diff-buffer-fn, 2-16
diff-input-from-artifact-contents-*type*, *2-15*
diff-input-from-artifact-contents-*type*, 2-16
display form, *2-6*
display-artifact-contents-*type*, *2-14*
display-artifact-contents-*type*, 2-17
display-artifact-reference-*type*, *2-14*
display-draft-reference-*type*, *2-17*
display-template-*type*, 2-21
dossier, 4-9, 4-10, 4-16, 4-27, 4-33
dossier-body, 4-27
dossier-contents, 4-27
dossier-reference-h-k-d, 4-27
dossier-requisite-k-d, 4-27
draft-contents- rom-artifact-contents-*type*, *2-17*
draft-contents-from-artifact-contents-*type*, 2-20
draft-edit-mode-*type*, *2-18*
draft-edit-mode-*type*, 2-19, 2-21

E_L_REPOSITORY, 4-21
edit-artifact, 2-17
elw (file), 4-35
elw:main, 4-32
enter-deriver, *4-8*, 4-31. *4-32*
error, 2-20
error (kind), 2-22, 2-23, 3-9, 4-11
eval-when-startup, 2-2, *2-5*
examine-artifact, 2-23
examine-derivative, 3-1, 3-2, 3-4. 3-8
examine-derivative-*derivative-class*, *3-3*
examine-error-derivative-*type*, *2-22*
examine-error-derivative-*type*, 2-23
examine-problems, 4-12
example-derive, *4-33*
executable-file, 4-4
export-derivative, 3-2
extenders-init, 4-31, *4-33*
extenders_init, 4-14

file, *2-16*
filename, 4-28
filename-and-uid, *4-13*, 4-14, 4-19, *4-34*
filename_and_uid, *4-18*. 4-19
Files

basic, 4-32
elw, 4-35
fixnum, 4-28
follow-link, 2-23
follow-link-hook, 2-23
future (plex), 1-2, 4-1–4-3

generic-pointer, 4-11
get-abort-job-data, *4-11*
get-and-delete-line, 2-22
getwd, 4-19

h-file (kind), 3-9
h-k, 4-28
h-k-d, 4-28
handle, *2-3*, 4-28, 4-34
handle-kind-derivative-sequence, 4-9, 4-16, 4-27, 4-28, 4-33
handle-kind-sequence, 4-9, 4-16, 4-28, 4-34
header-end-point, 2-7
hint, 4-10, 4-16, 4-17
hint-set, 4-10, 4-17, 4-28, 4-34
host, 4-3, 4-4
host default-job-server-class-variant, 4-4
hosts (plex), 4-3, 4-5, 4-35

id, 4-1
idle, 4-1, 4-3
index of a type, *2-1*
inhibition, *3-4*
insert-artifact-reference-;, 2-18
insert-artifact-reference-*type*, *2-16*
insert-error-derivative-contents, *2-23*
insert-handle-*type*, *2-18*
insert-two-artifact-references-*type*, *2-16*
integer, 4-17, 4-28, 4-34

job, *1-2*
job server, *1-2*, 4-1
job-server-class, 4-1–4-3

k-d, 4-28
k-d-set, 4-28
k-ur, 4-28
k_ur_seqh_make_gc, *4-21*
kill-server, 4-31
kill_server(), 4-24
KIND, *4-21*
kind, *3-1*, 4-8, 4-28, 4-34
kind-derivative-sequence, 4-9, 4-17, 4-27, 4-28
kind-derivative-set-sequence, 4-10, 4-17, 4-28, 4-34
kind-re, 4-6, 4-7
kind-re-replacement, 4-6
kind-sequence, 4-9, 4-11, 4-17, 4-28
kind-set, 4-2
kind/uid-request-sequence, 4-8, 4-10, 4-11, 4-17, 4-28, 4-34
kinds (of derived information)
bibliography, A-4
c-file, 3-9, 4-9
c-text, 3-2, 4-9
caption, A-1–A-4
error, 2-22, 2-23, 3-9, 4-11
h-file, 3-9
latex-directory, A-3, A-4
manuscript, 3-1, 3-9, 4-7, 4-20, A-1–A-5
standard-error, A-5
standard-output, A-5

latex-directory (kind), A-3, A-4
latex-piece (type), A-2
latex-root (type), A-3
lisp-source (type), 2-12
list, 4-27, 4-28
location, 4-8, 4-9, 4-17, 4-28, 4-34

M-x
cleanup-servers, 4-35
init, 4-35
kill-server, 4-35
load-file, 4-35
start-server, 4-35
stop-server, 4-35
M-x abort-job, 4-11
machine-type, 4-3, 4-4

manuscript (kind), 3-1, 3-9, 4-7, 4-20, A-1–A-5
merge-artifact-into-draft-*type*, *2-20*
merge-header-into-draft-*type*, 2-21

new-artifact-uid-*type*, *2-13*
new-derivative-class, *3-3*
new-deriver, *4-7*
new-draft-body-*type*, *2-17*
new-host, *4-4*
new-type, *2-1*, *2-2*, 2-2
new-worker, *4-5*
new_derivative_filename, *4-19*
nil, 4-27
non-deriver, *1-1*, *1-2*

partially bad derivative, 4-12
past (plex), 1-2
path, 4-28
pfasl (derivative class), 4-27
plexes
  artifacts, 2-2
  derivative-classes, 3-2–3-4, 4-12, 4-22, A-1
  derivers, 1-2, 1-3, 4-2, 4-3, 4-5–4-8, 4-22, 4-34, A-1
  future, 1-2, 4-1–4-3
  hosts, 4-3, 4-5, 4-35
  past, 1-2
  present, 1-2, 4-1–4-3, 4-23, 4-24
  types, 2-1, 2-2, 3-2, 3-3, 4-7
  workers, 1-2, 1-3, 4-2–4-5, 4-22–4-24, 4-30, 4-34, 4-35
present (plex), 1-2, 4-1–4-3, 4-23, 4-24
priority, 4-1
provide, 2-1

read-artifact-contents-*type*, *2-13*
read-artifact-contents-*type*, 2-14, 2-20
reference-handle-kinds, 4-9, 4-13
related_references, *4-20*
relative, *1-3*, *4-7*
repair-derivatives-class, *3-3*
repair-deriver, *4-7*
repair-host, *4-4*
repair-type, *2-2*, 2-5
repair-worker, *4-5*
repository form, *2-6*
requested-kind-sequence, 4-8, 4-9
require, 2-4
require-derivative-class, *3-3*
require-emacs-lisp, *2-4*
require-type, *2-4*
requisite-kinds, 4-9
residual-body, *2-11*
resource_path, *4-21*
revert-buffer, 2-21
revert-button-hooks, 2-21
revert-draft-source(, *2-22*
revert-draft-*type*, *2-21*
RPC signatures
  SIGINT, 4-35
RPCs
  start-server, 4-24
  stop-server, 4-24
  stop-worker, 4-24

same_file_contents, *4-20*, 4-21
scheduling plexes, *1-2*
select_derivatives, *4-21*
set-abort-job-data, 4-11, *4-11*, 4-11
shellscript (type), 4-7, A-5
SIGINT (RPC signature), 4-35
singleton_derivative_sequence, 4-16, *4-19*
source (type), 2-12
special-char-strings, 2-14
species, *3-1*
standard artifact text, *2-6*
standard-accept-handle-from-artifact-buffer, *2-15*
standard-accept-handle-from-draft-buffer, *2-18*
standard-artifact-body-uid-re, *2-9*
standard-artifact-commands, *2-15*

standard-artifact-contents-from-draft-contents, *2-20*
standard-artifact-contents-from-draft-contents-lock-list, *2-19*, 2-20
standard-artifact-edit-mode, *2-6*
standard-artifact-uid-from-draft-contents, *2-20*
standard-auto-save-draft, 2-21, *2-21*
standard-body-format-&-handles-that-appear-in-draft, *2-19*
standard-body-format-&-handles-that-appear-in-header, *2-11*
standard-create-draft, *2-17*
standard-delete-artifact-uid, *2-13*
standard-derivative-kinds, *2-7*
standard-diff-artifacts, *2-16*, 2-16
standard-diff-input-from-artifact-contents, *2-15*
standard-display-artifact-contents, 2-11, *2-14*
standard-display-artifact-reference, *2-14*
standard-display-draft-reference, *2-18*
standard-draft-contents-from-artifact-contents, *2-17*
standard-draft-edit-mode, *2-18*, 2-20
standard-error (kind), A-5
standard-examine-derivative, *3-3*, 3-5
standard-*f*, *2-5*
standard-handles-that-appear-in-draft, 2-19
standard-insert-artifact-reference, *2-16*
standard-insert-handle, *2-18*
standard-insert-two-artifact-references, *2-16*
standard-merge-artifact-into-draft, *2-20*, 2-21
standard-merge-header-into-draft, *2-21*
standard-new-artifact-uid, *2-13*
standard-new-artifact-uid-from-contents, *2-20*
standard-new-draft-body, *2-17*
standard-output (kind), A-5
standard-read-artifact-contents, *2-13*
standard-revert-draft, *2-21*, 2-21
standard-uid-of-artifact-body-re-format, *2-9*
standard-write-artifact-contents, *2-13*
start-server (RPC), 4-24
sth_and_uid, 4-14, 4-18, 4-19, *4-19*
stop-server, 4-31
stop-server (RPC), 4-24
stop-worker (RPC), 4-24
stop_server_receiver, 4-24
stream, 4-29
stream-and-uid, *4-13*, 4-13, 4-14, 4-18, *4-19*, 4-19, 4-30
stream_and_uid, 4-14, *4-18*, 4-19
string, 4-27- 4-30
string_sequence2_type, 4-16
subreference, *2-16*

t, 4-27
takeover-draft, 2-21
templated-artifact-contents-from-draft-contents, *2-20*
templated-body-format-&-handles-that-appear-draft, *2-20*
templated-body-format-&-handles-

that-appear-in-draft, *2-19*

templated-create-draft, *2-17*

templated-display-artifact-contents, *2-14*

templated-display-to-repository-lisp-source, *2-12*

templated-display-to-repository-module, *2-12*

templated-display-to-repository-*type*, *2-11*

templated-merge-artifact-into-draft, *2-21*

templated-repository-to-display, *2-12*

templated-repository-to-display-module, *2-12*

templated-repository-to-display-*type*, *2-11*

templated-repository-to-display-*type*, 2-14, 2-17

templated-revert-draft, *2-21*

timestamp, 4-1

tool, *1-2*, 4-2, 4-3, 4-5, 4-7

tool-list, 4-5

transpose-derivers, *4-7*

TYPE, *4-21*

type, 4-2

types

 area_type, *4-21*

 c-module, 4-6

 C-source, 4-14

 c-source, 3-2, 3-9, 4-9

 clisp-module, 4-25

 clisp-program, 4-25

 latex-piece, A-2

 latex-root, A-3

 lisp-source, 2-12

 shellscript, 4-7, A-5

 source, 2-12

 universal, 3-10, 3-11, 4-15, 4-18

 Unix-program, 4-14, 4-32

types (plex), 2-1, 2-2, 3-2, 3-3, 4-7

uid, *2-6*, *3-2*

uid-count, 4-8

uid-of-artifact-body-re-format-module, *2-12*

uid-of-artifact-body-re-format-*type*, *2-9*

uid-of-artifact-body-re-format-*type*, 2-9–2-11, 2-19

uids, 4-17, 4-28, 4-34

uniform-handle-kinds, *4-13*, A-1

uniform_handle_kinds, *4-18*

universal (type), 3-10, 3-11, 4-15, 4-18

universal-deriver, A-1

universal_deriver, A-1

Unix-program (type), 4-14, 4-32

use-templates, 2-11, 2-12

USER, 4-26

usual-start-server, 4-31

variant, 4-22

with-open-file, 4-30

worker, *1-2*, 4-2, 4-3, 4-5

workers (plex), 1-2, 1-3, 4-2–4-5, 4-22–4-24, 4-30, 4-34, 4-35

write-artifact-contents-*type*, *2-13*

yank-hooks, 2-18, 2-20